Oracle CDD/Repository™

Callable Interface Manual

Version 5.0

Reprinted 1995

Part No. A24846-2

ORACLE®

Oracle CDD/Repository Callable Interface Manual

Version 5.0

Part No. A24846-2

Contents

3 Storing and Manipulating Data

4 Working with Elements

7 Using Notices

8 Routine Descriptions

C Buffers

D Protocol Validations

E Literal Values

Index

Examples

Figures

Tables

Send Us Your Comments

Oracle Corporation welcomes your comments and suggestions on the quality and usefulness of this publication. Your input is an important part of the information used for revision.

You can send comments to us in the following ways:

- electronic mail — nedc_doc@us.oracle.com
- FAX — 603-897-3334 Attn: Oracle CDD/Repository Documentation
- postal service

  Oracle Corporation
  Oracle CDD/Repository Documentation
  One Oracle Drive
  Nashua, NH 03062
  USA

If you like, you can use the following questionnaire to give us feedback.

Name _____ Title _____

Company _____ Department _____

Mailing Address _____ Telephone Number _____

Book Title _____ Version Number _____

- Did you find any errors?
- Is the information clearly presented?
- Do you need more information? If so, where?
- Are the examples correct? Do you need more examples?

- What features did you like most about this manual?

If you find any errors or have any other suggestions for improvement, please indicate the chapter, section, and page number (if available).

Preface

This manual explains how to use the Oracle CDD/Repository callable interface. It also contains reference information for the callable interface.

Intended Audience

This manual is intended for the systems or application programmer who writes programs that directly invoke the object-oriented callable interface to Oracle CDD/Repository. You should be familiar with the architecture and capabilities of Oracle CDD/Repository before reading this guide.

Document Structure

This manual contains the following chapters and appendixes:

- Chapter 1 contains background information you should know before you write a program that calls the Oracle CDD/Repository callable interface.
- Chapter 2 describes how to get started with the callable interface.
- Chapter 3 describes how to store and manipulate data.
- Chapter 4 explains how to define new instances of an existing element type and how to create new element types.
- Chapter 5 explains how to send messages to the objects in a repository, describes method refinement and validation, and explains how to apply method refinement to the various categories of methods you can define.
- Chapter 6 explains how to define new properties.
- Chapter 7 describes notice services, notice-processing calls, and the NOTICE data type.
- Chapter 8 contains the reference descriptions for the callable interface routines.
- Appendix A describes error handling techniques.

- Appendix B describes the support utility routines.
- Appendix C describes the format of the buffers available through the callable interface.
- Appendix D summarizes the rules Oracle CDD/Repository enforces when you store record and field definitions in a repository.
- Appendix E lists Oracle CDD/Repository literal constants and their values.

Related Documents

Documents related to Oracle CDD/Repository include the following:

- *Oracle CDD/Repository CDO Reference Manual*
- *Using Oracle CDD/Repository on OpenVMS Systems*
- *Oracle CDD/Repository Architecture Manual*
- *Oracle CDD/Repository Information Model Volume I*
- *Oracle CDD/Repository Information Model Volume II*
- *Installing Oracle CDD/Repository on OpenVMS Systems*
- *Read Before Installing or Using Oracle CDD/Repository on OpenVMS VAX Systems* or, depending on your system, *Read Before Installing or Using Oracle CDD/Repository on OpenVMS Alpha Systems*

See online help for a glossary of defined terms.

Conventions

This manual uses the name Oracle CDD/Repository to refer to all versions of this product. Prior to Version 5.0, this product was known as CDD and CDD/Plus.

Table 1 shows the other conventions used in this manual.

Table 1 Documentation Conventions

Convention	Description
{ }	In format descriptions, braces indicate required elements. You must choose one of the elements.

(continued on next page)

Table 1 (Cont.) Documentation Conventions

Convention	Description
[ ]	In format descriptions, brackets indicate optional elements. You can choose none, one, or all of the options. (Brackets are not optional, however, in the syntax of a directory name in an OpenVMS file specification.)
( )	In format descriptions, parentheses delimit the parameter or argument list.
. . .	In format descriptions, horizontal ellipsis points indicate one of the following: • an item that is repeated • an omission, such as additional optional arguments • additional parameters, values, or other information that you can enter
. . .	Vertical ellipsis points indicate the omission of information from an example or command format. The information is omitted because it is not important to the topic being discussed.
italic type	Italic type emphasizes important information, indicates variables, and indicates complete titles of manuals.
boldface type	Boldface type in examples indicates user input. Boldface type in text indicates the first instance of terms defined either in the text, in the glossary, or both.
n.nn	A period in numerals signals the decimal point indicator. For example, *1.75* equals *one and three-fourths*.
UPPERCASE	Words in uppercase indicate a command, the name of a file, the name of a file protection code, or an abbreviation for a system privilege.
lowercase	In format descriptions, words in lowercase indicate parameters or arguments to be specified by the user.
$	A dollar sign ($) represents the OpenVMS DCL system prompt.
monospaced	This typeface indicates the name of a command, partition, path name, directory, or file. This typeface is also used in interactive examples and other screen displays.
NAMED_ELEMENT	The names of element types are set in small capitals.

(continued on next page)

Table 1 (Cont.) Documentation Conventions

Convention	Description
merge **hasChildren**	The names of messages and properties are set in bold type.
mod_1(2:a:3)	The names of repository elements are set in bold type.
list_value	References to values you supply, such as arguments, are set in italics. Names of routines also are set in italics.
"MCS_METHOD_ILLEGAL"	Property values are enclosed in double quotation marks.
0 ELEMENT 1 NAMED_ELEMENT 2 VERSION	The supertype–subtype relationship between element types is represented by numbers and by indentation. The example indicates that NAMED_ELEMENT is a supertype of VERSION and a subtype of ELEMENT.

1
Programming Concepts

This chapter contains background information you should know before you write a program that calls the Oracle CDD/Repository callable interface. The following topics are included:

- Oracle CDD/Repository operations
- binding styles
- symbol and entry point definition files
- routine entry points
- status return values
- data types
- numeric constants
- string-valued name arguments
- linking programs that call the Oracle CDD/Repository callable interface
- linking routines that are called by the Oracle CDD/Repository callable interface

1.1 Oracle CDD/Repository Operations

The Oracle CDD/Repository callable interface is an object-oriented interface that provides data integrity. Through this interface, your application calls procedures that send messages that affect repository elements. The Oracle CDD/Repository defines many data types and provides routines that operate on variables of these types. These routines allow the tool to be independent of the operating system.

The names of the calls and data types begin with MCS. These letters stand for Management Control System. The names, calling sequence, and effects of the routines are defined by ATIS (A Tools Integration Standard), to which the

Oracle CDD/Repository callable interface conforms. ATIS also defines the data types.

1.1.1 Order of Processing

There is a general order in which your program should call different groups of Oracle CDD/Repository routines. For example, your program must start a repository session before it can do anything else. Operations such as handling errors and dispatching to client functions can be done at any time.

Also, some calls can be made only after other calls. For example, because setting or reading a property value requires an argument list, you must build that argument list first. Similarly, you must allocate the value structures for the argument lists before you invoke the argument list calls. (Refer to Chapter 2 for more information on the order of processing.)

1.1.2 Related Routines

The following list outlines the groups of related callable routines:

- session management

  The Oracle CDD/Repository callable interface provides routines that allow you to start a session and make calls to the Oracle CDD/Repository callable interface that succeed or fail as a group. This group of calls is known as a **transaction**. A session may contain one or more transactions. Chapter 2 provides additional information on session management calls.

- typed value

  To ensure that programs can manipulate data regardless of how the underlying operating system represents that data, the Oracle CDD/Repository callable interface defines a set of opaque (you cannot see their implementation) data types and provides a set of routines to operate on those types. Refer to Chapter 3 for additional information on typed value calls.

- list manipulation

  A list is an ordered collection of value structures. Each list must have the data type MCS_LIST. Refer to Chapter 3 for additional information on list manipulation calls.

- argument list manipulation

  An argument list is a list that contains named arguments. If your program sends a message to an element, it frequently must supply arguments packaged in such an argument list. Chapter 3 explains how to use argument lists.

- element identifier (ID)

  An element ID uniquely identifies an element in the repository. Oracle CDD/Repository provides several types of routines to operate on element IDs.

- scan

  A scan is a data type that represents collections of element identifiers. It does not exist as an object in the repository; rather, its value is a set of the element identifiers that are related to the element whose property has a scan value. Chapter 3 explains how to use scans.

- message dispatch

  Your program sends messages to elements by calling *MCS_dispatch_op*. The routine takes an element ID, a message name, and an argument list that contains the arguments that message needs.

  You can refine a method to call *MCS_dispatch_superOp*, which sends the message to the supertype of the specified type.

  Chapter 5 details how to invoke *MCS_dispatch_op*; it also explains how to use *MCS_dispatch_superOp* when you write methods.

- notice services

  Oracle CDD/Repository provides **notices** to let programs and other clients know when definitions on which they depend have been changed. Chapter 7 tells how to use the Oracle CDD/Repository notice-passing mechanism.

- error handling

  Oracle CDD/Repository provides a set of calls to maintain and use the error stack. These calls are useful if you write your own methods or refine the methods supplied with Oracle CDD/Repository. Appendix A explains how to use the error stack routines.

Note

For better performance, always use MCS properties. Do not use CDD$ properties.

1.2 C and OpenVMS Bindings

The Oracle CDD/Repository callable interface defines the following:

- entry points for Oracle CDD/Repository callable interface routines
- data types
- symbolic constants (used to specify arguments and check status returns)

These definitions are supplied in the following forms:

- C bindings, which define entry points, data types, and symbols that follow UNIX C conventions. C bindings are listed in the file MCSPUB.H.
- OpenVMS bindings, which define entry points, data types, and symbols that follow OpenVMS conventions. Entry points are provided for most OpenVMS languages. These definitions are easiest to use from OpenVMS languages, but are less portable to environments other than OpenVMS. OpenVMS bindings are in files named MCS_PUB. File types are determined by the language.

1.3 Symbol and Entry Point Definition Files

Oracle CDD/Repository includes a global definitions file for several languages (see Table 1–1). When you compile your program with these definitions, the right data structures for your language are included in the program.

Table 1–1 Data Definition Files

File	Language Supported
MCSPUB.H	Portable C bindings
MCS_PUB_.ADA	Ada
MCS_PUB.PLI	PLI
MCS_PUB.MAR	MACRO
MCS_PUB.R32	BLISS (you must generate the .L32 file yourself)
MCS_PUB.H	DEC C/OpenVMS
MCS_PUB.FOR	DEC Fortran/OpenVMS
MCS_PUB.PAS MCS_PUB.ENV	DEC Pascal environment files

Note

MCS_PUB.H contains definitions for DEC C that use the OpenVMS binding style instead of the C binding style.

The files in Table 1–1 are located in SYS$LIBRARY.

Definitions for return status codes are contained in files called MCSMSG.H (for the C bindings) and MCS_MSG.* (for the OpenVMS bindings in the various languages). See Section 1.5 for information on how to use these codes. These files are in the same location (SYS$LIBRARY) as the MCS_PUB files.

Your application should use the appropriate language-specific mechanism to include these files in a compilation.

1.4 Routine Entry Points

The routine definitions differ between the C and the OpenVMS bindings in the following ways:

- names

  C bindings for the routines include only alphabetic characters and underscores: for example, MCS_list_getSize. The OpenVMS bindings follow OpenVMS conventions by including a dollar sign: MCS$list_getSize.

- string passing mechanisms

  C bindings expect null-terminated (ASCIZ) strings. OpenVMS bindings expect an OpenVMS string descriptor. (For details about string passing mechanisms, especially for strings whose value is returned by Oracle CDD/Repository, see Chapter 3.)

- scalar data type passing mechanisms

  C bindings expect small scalar data types (for example, MCS_LONGINT or MCS_BOOLEAN) to be passed by value. The OpenVMS bindings expect all data types to be passed by reference or by descriptor.

The descriptions of the routines in Chapter 8 show both the C binding and the OpenVMS binding. The C binding is given in the form of a function prototype. The OpenVMS binding is given by using standard OpenVMS entry point notation. You can examine the appropriate MCS_PUB file to see how the OpenVMS bindings translate to a particular language.

1.5 Status Return Values

All Oracle CDD/Repository callable interface routines return a status code whose value is a longword integer. Oracle CDD/Repository defines symbolic names and equates them with these codes. The name of each status code is listed with the description of the routine in Chapter 8. The symbolic name you use varies depending on the binding you use:

- For the C binding, the name of the status code is preceded with MCS_. For example, to test for ARGNOTFOUND, you use the symbol MCS_ ARGNOTFOUND.
- For the OpenVMS binding, the name of the status code is preceded with MCS$_, for example, MCS$_ARGNOTFOUND.

You must include the appropriate file (MCSMSG.H or MCS_MSG.*) in your compilation to define these symbols (see Section 1.3).

1.6 Data Types

There are two ways to identify an Oracle CDD/Repository data type:

- By a data type definition in languages that support user-defined data type definitions. An example is MCS_STATUS (C binding) or MCS$L_STATUS (OpenVMS binding). Use these definitions in data type declarations.
- By a global variable associated with a character string that is the name of the data type. An example is MCS_datatype_elementid (C binding) or MCS$r_datatype_elementid (OpenVMS binding). Some Oracle CDD/Repository callable interface routines require a string-valued argument that is the name of a data type. Use these variables instead of the name of the type because the variables remain the same even if the names change.

  Not all data types have global variables associated with their names because not all data types can be used as string-valued arguments to Oracle CDD/Repository callable interface routines.

Table 3–1 lists the Oracle CDD/Repository data types and shows the definitions and global variables associated with each.

1.7 Numeric Constants

Some Oracle CDD/Repository callable interface routines take an integer argument whose value indicates an action to be taken. An example is MCS_datatype_new, whose final argument indicates if you want to make a copy of the data. Two values are defined for this argument in the C binding: MCS_DATATYPE_ISCOPY and MCS_DATATYPE_NOTCOPY. The OpenVMS binding values are MCS$K_DATATYPE_ISCOPY and MCS$K_DATATYPE_NOTCOPY, respectively. Some properties also take on integer constant values or values that are the logical OR of several integer constants. The descriptions of these properties list the possible values.

You should use the defined symbolic constants, rather than the equivalent integers, if you set or test a value. Use the symbolic constants to ensure compatibility between versions of Oracle CDD/Repository.

1.8 String-Valued Name Arguments

Oracle CDD/Repository identifies several different types of entities by name. An example that has already been discussed is the data type. To specify a name argument, you must provide a null-terminated string (if using the C bindings) or an OpenVMS string descriptor (if using the OpenVMS bindings) whose value is the name of the entity.

Oracle CDD/Repository defines global variables whose values are the appropriate names for these entities. If possible, you should use these variables instead of the names to ensure compatibility with new versions of Oracle CDD/Repository.

Note

If you use the OpenVMS bindings on an OpenVMS system with any language other than DEC C, you can use any combination of uppercase and lowercase letters for OpenVMS bindings. If you use the OpenVMS bindings from DEC C on an OpenVMS system, or the C bindings on any system, type the name exactly as shown (for example, MCS_prop_hasChildren).

Global variables are defined for the following types of names:

- message names

  An example is **MCS_message_open** (C binding) or **MCS$r_message_open** (OpenVMS binding). Message names are arguments to the *MCS_dispatch_op* and *MCS_dispatch_superOp* routines.

- property names

  An example is **MCS_prop_hasChildren** (C binding) or **MCS$r_prop_ hasChildren** (OpenVMS binding). Property names are in property lists sent with the GETPROP, SETPROP, and NEW messages.

- element type names

  An example is MCS_ELM_VERSION (C binding) or MCS$R_ELM_VERSION (OpenVMS binding).

- argument names

  Oracle CDD/Repository accepts message arguments in nonpositional argument lists, with each argument identified by name. Use the global variables to specify the names of the arguments. An example is *MCS_arg_ new_inst_elmID* (C binding) or *MCS$r_arg_new_inst_elmID* (OpenVMS binding).

The description of each message, property, element type, and argument lists the global variables associated with its name in each binding. These descriptions are listed in Chapter 8.

1.9 Linking Applications That Call Oracle CDD/Repository

Applications that call Oracle CDD/Repository routines must be linked with the shareable image called CDDSHR.EXE and the run-time library called MCSRTL.OLB. These files are placed in the SYS$LIBRARY and SYS$SHARE directories when Oracle CDD/Repository is installed on your system.

The following example shows how to link an object file called REPODOC.OBJ.

The source file for REPODOC.OBJ is a C program. Therefore, the application is also linked with the C shareable image called VAXCRTL.EXE.

```
$ link repodoc, sys$input/opt
sys$library:mcsrtl/lib
sys$share:cddshr/share
sys$share:vaxcrtl/share
Ctrl/Z
```

You can put the library and shareable image specifications in a linker options file. Then you do not have to enter them at the keyboard. For example, an options file for linking REPODOC.OBJ would contain the following lines:

```
sys$library:mcsrtl/lib
sys$share:cddshr/share
sys$share:vaxcrtl/share
```

If this file is called REPOS.OPT, then REPODOC.OBJ can be linked with the command:

```
$ link repodoc, repos/opt
```

1.10 Linking Applications Called by Oracle CDD/Repository

Methods that are called by Oracle CDD/Repository are external code methods. External code methods are entry points in a shareable image. Therefore, to link an application called by Oracle CDD/Repository, you must link the methods into a shareable image and place the image in an appropriate directory. (See Chapter 5 for information on how to write external code methods.)

2
Getting Started

This chapter explains how to get started with an Oracle CDD/Repository application and provides overviews of typical application tasks.

You should thoroughly understand the concepts explained in the *Oracle CDD/Repository Architecture Manual* as they pertain to your application. Chapter 8 of this manual provides detailed descriptions of the Oracle CDD/Repository callable interface routines. The *Oracle CDD/Repository Information Model Volume I* and *Oracle CDD/Repository Information Model Volume II* manuals contain reference information on the Oracle CDD/Repository type hierarchy.

2.1 Basic Operations

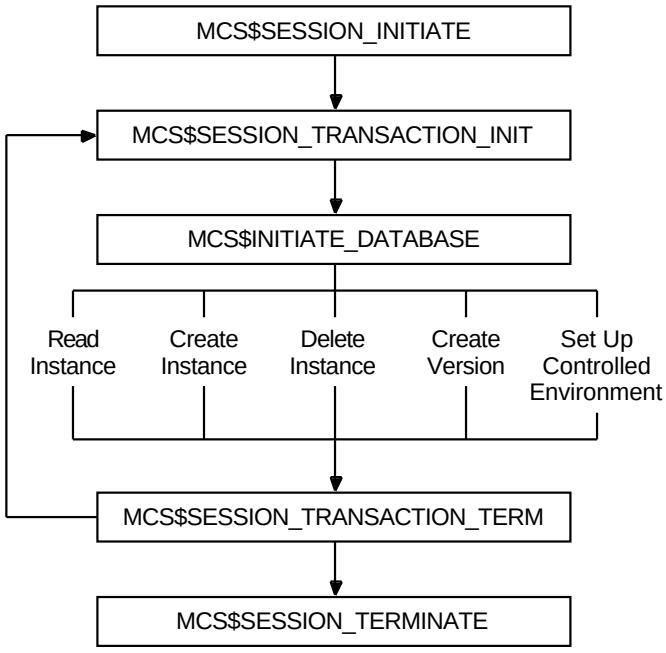
Using the Oracle CDD/Repository callable interface involves four basic activities:

- beginning and ending sessions
- starting and ending transactions
- opening and closing repositories
- manipulating the repository

Figure 2–1 shows the programming flow of Oracle CDD/Repository callable interface routines and the following typical repository manipulation tasks:

- reading an instance of an existing element
- creating a new instance of an existing element
- deleting an instance of an existing element
- creating a new element
- setting up a controlled environment for systems development

Figure 2–1 Programming Sequence



ZK–3566A–GE

2.2 Using Sessions

A **session** is the context in which your application does its work. Figure 2–1 shows that your application can have only one repository session active at a time, but that session can include many transactions (only one transaction can be active at a time).

Session processing includes the following functions:

- defining a session handle

  The repository system identifies your session by assigning a unique value to its handle. A session **handle** is a data structure that identifies a specific session. Various Oracle CDD/Repository routines require the handle as an input argument.

  Your C program declares a variable for the session handle by specifying a data type of MCS_SESSION. The corresponding OpenVMS type definition is MCS$R_SESSION.

- starting a session

  To start a session, your program calls *MCS_session_initiate* with one parameter, the session handle. Oracle CDD/Repository starts a session and assigns a handle, which it returns to your program in the session parameter.

  All subsequent calls to start or end a transaction, to access a repository, or to end the session must include this session handle. Your program can have only one session open at a time. The session can include many transactions and each transaction can open a new repository. More than one session can access each repository.

- ending a session

  When your program is done working in the repository environment, it calls *MCS_session_terminate* to end the session and free the memory associated with it. Ending a session also closes access to all repository databases opened by transactions in that session.

  All element IDs assigned by the transactions within this session are made invalid; they cannot be reused unless they were made persistent by a call to *MCS_elmID_export_persistent*.

An element ID is valid only during the session in which it is first returned. To find the same element in another session, you either find it by name, or create a **persistent element ID** from the the element ID returned during the first session. A persistent element ID remains valid from one session to the next, but cannot be used in place of a regular element ID; you must convert it back to a regular element ID once you start the new session.

2.3 Using Transactions

A **transaction** is a group of repository operations that succeed or fail as a group. If one statement fails, the whole transaction fails. Figure 2–1 shows that your program must have a transaction started before it does any operation that accesses a repository, including opening the repository database. It also shows that you should perform only one application task against a given repository in a given transaction.

You cannot start a new transaction if a transaction is active.

If an operation in a transaction fails, you may need to remove the changes your program made to the repository database. This is called **rolling back** the transaction. When all operations in the transaction succeed, your program should make the changes permanent. This is called **committing** the transaction. You specify if you want the transaction committed or rolled back on the call to *MCS_session_transaction_term*.

If your transaction is read-only, you only need to end it. You need not be concerned with committing or rolling back a read-only transaction.

There are several Oracle CDD/Repository routines that can be used only in a transaction; for example, sending a message. Most operations involving element IDs also cannot execute unless a transaction is active.

Some applications need to manipulate files stored in the native file system at the same time they operate on repository elements. If done during a transaction, these manipulations should be canceled if the transaction aborts in order to keep the external files synchronized with the repository. A number of routines allow you to tie your external file manipulations into the Oracle CDD/Repository transaction control mechanism. Then, if a transaction aborts or the system fails during a transaction, Oracle CDD/Repository can automatically roll back the state of the external files at the same time it rolls back the repository.

Transaction processing includes the following functions:

- defining transaction handles

  The repository system identifies each transaction by assigning it a unique handle. A transaction handle is a data structure that identifies a specific transaction. Various Oracle CDD/Repository routines require the handle as an input argument.

  Your C program declares a variable for the transaction handle by specifying a data type of MCS_TRANSACTION. The corresponding OpenVMS type definition is MCS$R_TRANSACTION.

- starting transactions

  To start a transaction, your program calls *MCS_session_transaction_init*. This routine takes four parameters:

  - A transaction handle identifying an existing transaction. This handle must be zero; it is present to allow future support of nested transactions.
  - A transaction handle to receive the handle of the newly started transaction.
  - The handle of the session under which the transaction is started. This is the handle returned by a previous call to *MCS_session_initiate*.
  - A BOOLEAN flag specifying if the transaction is a read-only transaction.

- ending transactions

  MCS_session_transaction_term ends the transaction and sets the transaction handle back to zero.

  To make your program changes permanent:

  1. Specify the constant MCS_TRANSACTION_COMMIT (the same as setting the *abort_flag* parameter to "false").
  2. Call *MCS_session_transaction_term*, specifying the handle of the transaction you wish to terminate.

  If you want to delete (or roll back) the changes your program made, set the *abort-flag* parameter to MCS_TRANSACTION_ABORT.

  Note

  Dispatching too many messages in the same transaction may cause your system to run out of virtual memory. To prevent or correct this problem, you can either increase your page file quota or terminate transactions more often.

  Your system also can run out of virtual memory if you change a member of the type hierarchy, which can cause a *reserve-to-top* operation to modify a prohibitively large number of objects.

To support your transactions, there are a set of routines that allow you to perform journaled file operations through Oracle CDD/Repository, instead of through calls to the native file system. These routines include the following:

- *MCS_fileop_journal_create*—Indicates that the application created a file and provides for the file to be deleted by Oracle CDD/Repository during rollback.
- *MCS_fileop_delete*—Deletes the file and provides for it to be restored during rollback.
- *MCS_fileop_journal_modify*—Provides a reference point to which the file can be rolled back by Oracle CDD/Repository.

Other journaled file operations provide for file renaming, creation and deletion of symbolic links to files, and creation and deletion of file-system directories. Oracle CDD/Repository does these operations when you call the appropriate routine, and undoes them if you roll back the transaction. (See Chapter 8 for more information on journaled file routines.)

2.4 Accessing Repository Databases

Before you can operate on the contents of a repository, you must initiate (gain access to) it. This serves two purposes:

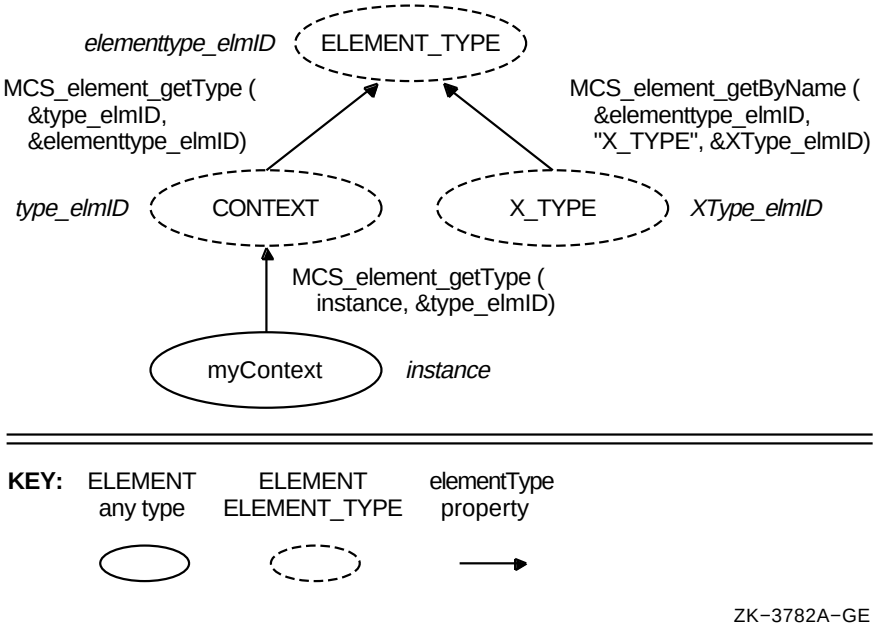
- It prepares the repository for use by your application.
- It returns to your application the root of the repository's **instantiation hierarchy**. The instantiation hierarchy is a tree in which children are instances of the type defined by their parent. Given the root of this tree, an application can find its way to any instance (element) in the repository.

You only need to initiate a database once during a session; however, there are performance considerations. If a database is accessed infrequently, you may want to initiate and close it for each transaction. If a database is accessed often, leave it open to avoid repeated calls to initiate the database.

Figure 2–2 shows how, starting with the element ID of any element, you can find the element ID of the ELEMENT_TYPE element that represents a specified type. The figure starts with a CONTEXT element but it could be an element of any type.

You first find the element type of CONTEXT by getting the element ID of MYCONTEXT. Next, you find the element type of ELEMENT_TYPE by getting the element ID of CONTEXT.

Figure 2–2 Finding an Element Type



To define a new physical repository, call the *MCS_DB_new* routine with the following information:

- specification of the file system directory in which you want the repository files created
- session ID

The file system directory must already exist, and it must be empty.

You also can define a repository through the Oracle CDD/Repository CDO interface.

To delete a physical repository, call the *MCS_DB_free* routine with the specification of the file system directory that holds the repository you want to delete. In response, Oracle CDD/Repository detaches the repository from its session and all element IDs associated with the repository become invalid.

If transactions are active when you call *MCS_DB_free*, the deletion fails.

2.5 Manipulating a Repository

After you have opened a repository in a transaction, you can manipulate the contents of that repository by sending messages to its elements. You send a message either to make a change in the repository state or to gather information about the repository state. You can change a repository state in one of the following ways:

- Send a message that performs a specific operation, such as RESERVE, REPLACE, or OPEN. These messages are implemented by methods that perform a number of low-level operations, including creating new elements and changing property values.
- Sending the SETPROP message to explicitly change the value of a property.

Note

For better performance, always use MCS property names. Do not use CDD$ property names. (See the *Oracle CDD/Repository Information Model Volume I* and *Oracle CDD/Repository Information Model Volume II* manuals for more information on properties.)

Gathering information about a repository typically involves sending the GETPROP message to return the value of one or more properties.

Sending a message involves a cycle of steps:

1. Determine the element ID of the element to which the message is to be sent. You can locate the element by name, or it may be a value set in an output argument of a previous routine.
2. Prepare an argument list to accompany the message. Different messages take different (required and optional) arguments. Several routines create and manipulate argument lists.
3. Send the message to the element by using the *MCS_dispatch_op* routine. The *MCS_dispatch_op* routine specifies the element ID to receive the message, the name of the message, and the argument list. When used with one of the dispatch routines, the argument list is called a **dispatch list**. The dispatch list is available to the method that responds to the message.

These also are the steps you use to read an instance of an element, as shown in Figure 2–1.

The next step is to analyze the results of the message. Oracle CDD/Repository presents message results in a number of ways:

- as a success or error status return from *MCS_dispatch_op*
- as values placed in output arguments by the methods that implement the message
- as status codes associated with individual arguments in the dispatch list
- as entries on the **error stack**, a global data structure upon which method routines (and the routines they call) place information about errors that occurred during message processing

Example 2–1 shows the pseudo code for the typical order of processing of the Oracle CDD/Repository routines you can use to perform steps 1 through 3. (Be sure to check the status returned for each call.)

Example 2–1 Reading an Instance

```
MCS$ELEMENT_GETBYNAME (element_type, ❶
                instance_type_name,
                instance_Type);

MCS$ELEMENT_GETBYNAME (instance_type, ❷
                  instance_name,
                  instance_elmId);

MCS$LIST_NEW (prop_list, ❸
                %REF(4),
                %REF(4));

MCS$ARGLIST_ADDARG (MCS$R_PROP_NAME, ❹
                0,
                %REF(1),
                prop_list);

MCS$LIST_NEW (disp_list, ❺
                %REF(4),
                %REF(4));

MCS$ARGLIST_ADDARG (mcs$r_arg_arglist, ❻
                prop_list,
                %REF(1),
                DISPLIST);

MCS$DISPATCH_OP (your_type_elmId, ❼
                mcs$r_message_GETPROP,
                disp_list);
```

(continued on next page)

Example 2–1 (Cont.) Reading an Instance

```
MCS$ARGLIST_FINDARG (disp_list, ❽
                mcs$r_arg_arglist,
                prop_list,
                status,
                index);

MCS$ARGLIST_FINDARG (prop_list,
                mcs$r_prop_yourProperty,
                atis_value,
                status,
                index);

MCS$DATATYPE_READ (atis_value, ❾
                data_type,
                your_value);
```

The following list describes the steps in Example 2–1 used to read an instance:

❶ Get the type of the instance.

❷ Get the instance.

For more information on finding an instance of an element, see Section 2.5.1.

❸ Build a property list.

❹ Build an argument specification and place it in an argument list. Because **getProp** sets output arguments, set the value to zero.

❺ Create a dispatch list.

❻ Insert the property list (use the *mcs$r_arg_arglist* argument) on the dispatch list.

For more information on preparing an argument list, see Section 2.5.2.

❼ Return the properties.

For more information on sending a message, see Section 2.5.3.

❽ Read the property list.

❾ Use this routine to read a simple data type.

Do *not* use *MCS_datatype_read* on a scan or a list. Use the processing shown in the following pseudo code example to read a scan.

```
WHILE .status nequ MCS$_ENDFIND
DO
   status = MCS$SCAN_GET_NEXT (scan, rel_elmId, inst_elmId)
```

Use the pseudo code shown in Example 2–2 to read a list. (Be sure to check the status returned for each call.)

Example 2–2 Reading a List

```
MCS$$LIST_GETSIZE (arglist, size) ❶
incr index from 1 to .size do
    BEGIN
    MCS$LIST_GET( list, ❷
                index,
                value);
```

❶ Before you read a list, get the size of the list.

❷ Use this routine to read the list.

Use the pseudo code shown in Example 2–3 to read an argument list. (Be sure to check the status returned for each call.)

Example 2–3 Reading an Argument List

```
MCS$$LIST_GETSIZE (arglist, size) ❶
incr index from 1 to .size do
    BEGIN
    MCS$$ARGLIST_GETARG(arglist, ❷
                index,
                value,
                status,
                name);
    END;
```

❶ As with the previous example, you must get the size of the list before you can read it.

❷ Use this routine to read the argument list.

For more information on returning information, see Section 2.5.4.

2.5.1 Reading an Instance of an Element

Before you can read the properties of an instance of an element, you must have its element ID. The following list describes several techniques to find element IDs. Many of these techniques use the results from previous messages as their starting point.

- Some Oracle CDD/Repository routines (other than those that send messages) set values for element IDs as output arguments. You can use these element IDs to find information about the corresponding elements.
- If you know the name and type of an element, you can use the *MCS_element_getByName* routine to find its element ID. (However, you specify the type in the form of an element ID, so you must already have obtained an element ID of the type before calling *MCS_element_getByName*. The initial call to *MCS_initiate_database* returns an element ID that provides access to all the element types in the repository.)
- Some element properties have single element IDs as their values. Use the GETPROP message (with the *MCS_dispatch_op* routine) to obtain this value.
- Other element properties have multiple element IDs as their values. (These are called scans.) You can find a single element ID in the scan by calling the *MCS_scan_getByName* routine. You also can use *MCS_scan_getNext* to iterate through the scan.
- Two other ways to obtain scans of element IDs are *MCS_scan_dir*, which returns a scan of elements whose names match a specified pattern, and *MCS_scan_query* for more general repository queries.
- Some methods return element IDs in their argument lists. For example, the methods for the **new** and **reserve** messages, both of which create elements, return the element ID of the created element in the *new_inst_elmID* argument. (See the *Oracle CDD/Repository Information Model Volume I* manual for more information on messages and methods.)

2.5.2 Preparing Argument Lists

Messages can take required or optional arguments. (The message descriptions in the *Oracle CDD/Repository Information Model Volume I* manual list and describe the arguments taken by each message.) The arguments supply information needed to carry out the operation or allow Oracle CDD/Repository to pass information back to the sender of the message.

Required arguments are essential to a message; the operation fails if they are not supplied. For example, the output argument *new_inst_elmID* is required for NEW and RESERVE, since otherwise Oracle CDD/Repository has no way to identify to the sender the element it creates. The EXPORT message requires an input argument, *fname*, to identify the file to create in the native file system.

Optional arguments supply supplementary information with the message. Typically, optional arguments specify processing options to override system defaults. For example, the RESERVE message by default creates a new version of an element that is the direct successor of the target of the message. An optional argument, *branch_name*, causes the new version to be created on a branch.

Note

You cannot create metadata elements on a branch.

Some messages take no explicit arguments. For these messages, the information contained in the element itself is sufficient for processing the operation.

You pass arguments with messages in the form of argument lists. An argument list is a specialized form of the MCS_LIST data type in which each list entry consists of an argspec. An **argspec** is a triple argument consisting of a name, a value, and a status field. (The status field is frequently not used.) The following routines manipulate argument lists:

- The *MCS_arglist_addArg* routine adds an argspec to an argument list.
- The *MCS_arglist_findArg* and *MCS_arglist_getArg* routines find argspecs in an argument list by name and by index, respectively. Use these to find arguments in an argument list that your application has been passed, or to retrieve return argument values from an argument list your application passed with a message.
- The *MCS_arglist_setNameValue* and *MCS_arglist_setIndexValue* routines modify argspecs specified by name and by index, respectively.

See Chapter 3 for more information on list and argument list routines.

2.5.3 Sending Messages

Sending a message to an element is simply a matter of calling the *MCS_dispatch_op* routine, specifying the element ID of the target, the name of the message, and the argument list (which may be omitted if the message takes no required arguments). The following section explains how to interpret the results of the call.

A related routine, *MCS_dispatch_superOp*, also sends a message but requests that the method provided by a supertype be invoked, instead of the method provided by the element's type. The *MCS_dispatch_superOp* routine is used when you write a method refinement that, at some point, needs to invoke the supertype's method. The arguments are the same as for *MCS_dispatch_op* with the addition of an argument that specifies the type whose supertype method you want to invoke. (See Chapter 5 for more information on these routines.)

2.5.4 Analyzing Results

The results of a routine appear in various forms. You should first determine whether the routine succeeded or failed (and take appropriate action), then collect and use the information returned by the routine.

2.5.4.1 Possible Errors

You should check the status return immediately after calling a routine. A call that succeeds returns a SUCCESS status. A SUCCESS status indicates that the operation completed successfully (you can continue to analyze the results as outlined in Section 2.5.4.2). A status other than SUCCESS indicates some kind of unexpected result, requiring further examination.

When you receive a status other than SUCCESS, there are three sources of information about what happened:

- error status

  There is a small set of status returns that indicate a result from which you can recover. For example, the GETPROP message may fail to retrieve all the property values you requested. You may choose to examine the list of properties to see which were not retrieved, and continue without those values.

  In general, however, a status other than SUCCESS (especially if you are dispatching a message) indicates a serious problem.

- error stack

  When an error occurs during message processing (or any other routine call), the method function that detects the error places an error status on the error stack, a global data structure. The first detection may occur deep in a calling structure; as the calling sequence unwinds, other method functions may push entries onto the error stack. When *MCS_dispatch_op* returns, the error stack can contain many entries. Typically, the root cause of the error is at the bottom of the stack, with higher entries representing the "interpretation" of the error by the method functions that placed them there.

  There are several Oracle CDD/Repository routines your application can call to examine the contents of the error stack, to manipulate entries, and to clear the error stack between routine calls.

- argument list

  For the SETPROP and GETPROP messages, Oracle CDD/Repository fills in the status field of the argspec with an error status when the operation on that property fails.

Based on your application's analysis of the error, it may choose to continue the transaction (after fixing up the results if necessary) or to abort the transaction. Aborting the transaction returns the repository to its state before the transaction started.

Note

There are certain *severe* errors that will not allow you to continue processing because it will corrupt the repository. For these errors, you only can abort the transaction. (See Chapter 8 for more information.)

2.5.4.2 Using Returned Information

There are two sources of information about a successful message dispatch:

- argument list

  Methods place output arguments on the argument list, from which they can be retrieved by calling the *MCS_arglist_findArg* or *MCS_arglist_getArg* routines. For the GETPROP message, Oracle CDD/Repository fills in the returned value in the argspec corresponding to the property.

- error stack

  Some messages place additional information about successful completion on the error stack. For example, the DIFFERENCES message indicates on the error stack whether or not differences were found in the two files it compares. In either case, the call to *MCS_dispatch_op* returns SUCCESS.

3

Storing and Manipulating Data

Oracle CDD/Repository stores information in the form of elements and properties. Property values may consist of data stored with the element, may express relationships between elements, or may be computed when the value is requested. An application gains access to information contained in properties by sending the GETPROP message. An application changes information by sending the SETPROP message.

This chapter explains why the data that makes up Oracle CDD/Repository information is typed and how value structures are used to process data. The chapter also describes how to use the various Oracle CDD/Repository data types.

3.1 Data Types

Data in Oracle CDD/Repository is typed. Properties contain data of a defined type, and routines accept and return arguments of defined types. The data types cover the familiar scalar types such as integer and floating point, as well as more complex and specialized types such as element IDs and scans, which are tailored to the needs of the Oracle CDD/Repository callable interface.

To allow data to be passed into and out of the repository without regard to its type, Oracle CDD/Repository provides value structures. A **value structure** is a means of packaging various types of data in a uniform manner for use in the repository. Routines exist to:

- store and free data in value structures
- compare the values they contain
- extract those values so they can be used by the application

Value structures and the routines that manipulate them make up an abstract mechanism for handling data. See Section 3.2 for information about value structures.

The Oracle CDD/Repository data types fall into two overlapping categories:

- data types that can be declared in programs

  For these types, Oracle CDD/Repository provides a data type definition in those languages that support user-defined types. Most data types can be declared, but a few, such as MCS_datatype_list (MCS$r_datatype_list is the OpenVMS symbolic name) and MCS_datatype_scan (MCS$r_datatype_scan is the OpenVMS symbolic name), cannot. For these types, you must declare a value structure to store the data, and pass the value structure to the Oracle CDD/Repository routines that manipulate the data.

- data types that can be stored in value structures

  Routines that store and retrieve data in value structures use the character-string name of the data type to specify the type. Oracle CDD/Repository provides global variables that correspond to the name of each data type that can be stored in a value structure.

Table 3–1 summarizes the Oracle CDD/Repository data types and indicates where you can find more information about each one.

Table 3–1 Oracle CDD/Repository Data Types

Data Type (C Binding)[1] Data Type (OpenVMS Binding)[1]	C Symbolic Name[2] OpenVMS Symbolic Name[2]	Description	See Section
MCS_BOOLEAN MCS$L_BOOLEAN	MCS_datatype_boolean MCS$r_datatype_boolean	A true/false value.	3.4
MCS_DOUBLE MCS$D_DOUBLE	MCS_datatype_double MCS$r_datatype_double	A double-precision floating-point number.	3.3
MCS_ELEMENTID MCS$R_ELEMENTID	MCS_datatype_elementid MCS$r_datatype_elementid	A handle to an element; an element ID.	3.8
MCS_FLOAT MCS$F_FLOAT	MCS_datatype_float MCS$r_datatype_float	A floating-point number.	3.3

[1]A symbolic constant that identifies the data type for use in declarations. If "None," then the data type cannot be declared in a program; it can only be stored in a value structure.

[2]A global variable whose value is a string that is the name of the data type. Use this variable to specify the name of the data type in calls to *MCS_datatype_new* and other routines that require it. If "None," then no variable is defined because the name of the data type is never used as an argument.

(continued on next page)

Table 3–1 (Cont.) Oracle CDD/Repository Data Types

Data Type (C Binding)[1] Data Type (OpenVMS Binding)[1]	C Symbolic Name[2] OpenVMS Symbolic Name[2]	Description	See Section
None	MCS_datatype_list MCS$r_datatype_list	An ordered list of value structures. The members of a list may be of different data types.	3.10
MCS_LONGINT MCS$L_LONGINT	MCS_datatype_longint MCS$r_datatype_longint	An integer stored in a longword.	3.3
MCS_MEMBLOCK MCS$A_MEMBLOCK	MCS_datatype_memblock MCS$r_datatype_memblock	A memory block of arbitrary size whose contents must be managed entirely by either the application or Oracle CDD/Repository.	3.7
MCS_VMSTIME MCS$Q_VMSTIME	MCS_datatype_vmstime MCS$r_datatype_vmstime	A representation of the date and time in OpenVMS format.	3.6
None	MCS_datatype_notice MCS$r_datatype_notice	A representation of a change notice sent to elements that depend on an element that changes. *This data type is an Oracle CDD/Repository extension to ATIS.*	7.7
None	MCS_datatype_scan MCS$r_datatype_scan	An unordered collection of element IDs.	3.9
MCS_SESSION MCS$R_SESSION	None	A session handle.	2.2
MCS_SMALLINT MCS$W_SMALLINT	MCS_datatype_smallint MCS$r_datatype_smallint	An integer stored in 16 bits.	3.3
MCS_STATUS MCS$L_STATUS	None	A status return value.	1.5
MCS_STRING MCS$A_STRING	MCS_datatype_string MCS$r_datatype_string	A null-terminated character string.	3.5
MCS_STRINGDSC MCS$R_STRINGDSC	MCS_datatype_stringdsc MCS$r_datatype_stringdsc	An OpenVMS string descriptor.	3.5

[1]A symbolic constant that identifies the data type for use in declarations. If "None," then the data type cannot be declared in a program; it can only be stored in a value structure.

[2]A global variable whose value is a string that is the name of the data type. Use this variable to specify the name of the data type in calls to *MCS_datatype_new* and other routines that require it. If "None," then no variable is defined because the name of the data type is never used as an argument.

(continued on next page)

Table 3–1 (Cont.) Oracle CDD/Repository Data Types

Data Type (C Binding)[1] Data Type (OpenVMS Binding)[1]	C Symbolic Name[2] OpenVMS Symbolic Name[2]	Description	See Section
MCS_TRANSACTION MCS$R_TRANSACTION	None	A transaction handle.	2.3
MCS_VALUE MCS$R_VALUE	None	A value structure; a structure that packages data for use in the repository. Many Oracle CDD/Repository routines accept input or output arguments in the form of value structures.	3.2

[1]A symbolic constant that identifies the data type for use in declarations. If "None," then the data type cannot be declared in a program; it can only be stored in a value structure.

[2]A global variable whose value is a string that is the name of the data type. Use this variable to specify the name of the data type in calls to *MCS_datatype_new* and other routines that require it. If "None," then no variable is defined because the name of the data type is never used as an argument.

Examine the symbol definition file (MCSPUB.H for the C bindings, or MCS_PUB.* for language-specific OpenVMS bindings) if you want to know the structural details of these data types.

Note

These structures may not remain the same for each version of Oracle CDD/Repository. To promote compatibility, use the symbolic constants provided to declare data, and use the Oracle CDD/Repository callable interface routines to manipulate data rather than accessing the structures directly.

3.2 Packaging Data in Value Structures

A value structure is an opaque (you cannot see its implementation) data structure that packages data for repository use. You can manipulate a value structure only by using the routines provided for that purpose.

A value structure can store many different types of data. Each instance of a value structure contains information about the data that it stores. The value structure may store the data within the structure (for the shorter data types) or store a pointer to the data.

The purpose of a value structure is to allow applications and the repository to pass data without knowing the type of the data. For example, if you send the GETPROP message, you supply a value structure to receive the property value; you do not have to supply a variable whose type matches that of the property value. If you send SETPROP, you package the new value in a value structure. This simplifies the construction and handling of argument lists by making them uniform and independent of the type of data that they contain.

A value structure has the data type MCS_VALUE (C binding) or MCS$R_VALUE (OpenVMS binding).

3.2.1 Loading Data in Value Structures

To initialize a value structure (typically, if you need to supply one as an input argument) use the *MCS_datatype_new* routine. This routine stores data that you supply in the value structure. You must specify the data type with *MCS_datatype_new*; the value structure also stores the data type.

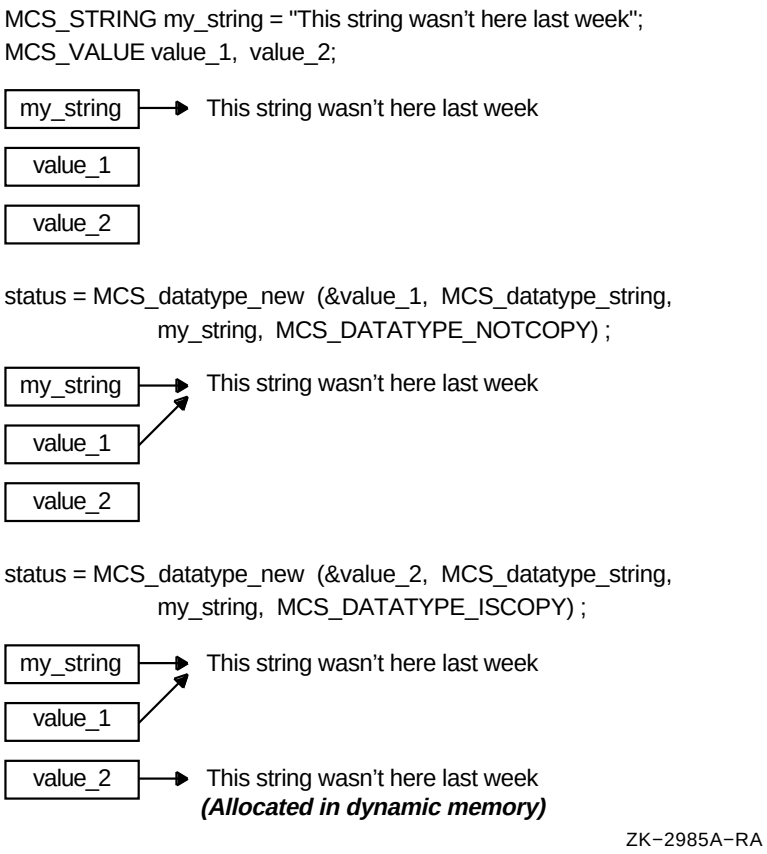
The following example creates two value structures. One contains an integer value, and one contains a string. (It is assumed that the value of *my_string* is supplied elsewhere in the code.)

```
MCS_VALUE   value_int, value_string;
MCS_LONGINT x;
MCS_STRING  my_string;     /* This is assumed to be passed in */
MCS_STATUS  status;
   .
   .
   .
    x = 10;
    status = MCS_datatype_new(
        &value_int,             /* The value structure */
        MCS_datatype_longint,   /* Name of type to store */
        &x,                     /* The value to store */
        MCS_DATATYPE_ISCOPY);   /* Copy flag, see below */
   .
   .
   .
    status = MCS_datatype_new(
        &value_string,
        MCS_datatype_string,
        my_string,
        MCS_DATATYPE_ISCOPY);
   .
   .
   .
```

You can use *MCS_datatype_new* to store values of most Oracle CDD/Repository data types in value structures. (For a complete list, see the description of *MCS_datatype_new* in Chapter 8.) There are two notable exceptions: lists and scans. These two data types have their own new routines, although you can delete them by using *MCS_datatype_free*, described later.

If you supply data with *MCS_datatype_new*, you also can specify if the value structure should reference the data you supply or a copy of the data. Take, for example, the value structure *value_string* in the preceding example. The fourth argument to the call that loads *value_string* is a flag that causes the data to be copied. If you use this flag, *MCS_datatype_new* creates a copy of the data (in this case, copies the string); the value structure points to this copy. If you specify MCS_DATATYPE_NOTCOPY instead, the value structure points to the string you supply. Figure 3–1 illustrates the distinction.

Figure 3–1 Storing Data in Value Structures



A value structure records if you specified ISCOPY or NOTCOPY with *MCS_datatype_new*. The behavior of *MCS_datatype_free*, which frees data pointed to by a value structure, depends on this setting. See Section 3.2.3 for more information.

3.2.2 Retrieving Data from Value Structures

The *MCS_datatype_read* routine copies data from a value structure into a form that can be used directly by an application. You can use *MCS_datatype_read* to retrieve values of any data type that can be declared.

Note

Do not use the *MCS_datatype_read* routine for value structures that contain nondeclarable types, such as lists (MCS_datatype_list), scans

(MCS_datatype_scan), or notices (MCS_datatype_notice). Use the routines provided for manipulating these data types.

The application is responsible for allocating sufficient memory to contain the data, and for freeing that memory when it is no longer needed. The *MCS_datatype_read* routine always copies the data from the memory associated with the value structure to the memory provided by the application. After *MCS_datatype_read* has copied the data from the value structure, there is no connection between the value structure and the returned data.

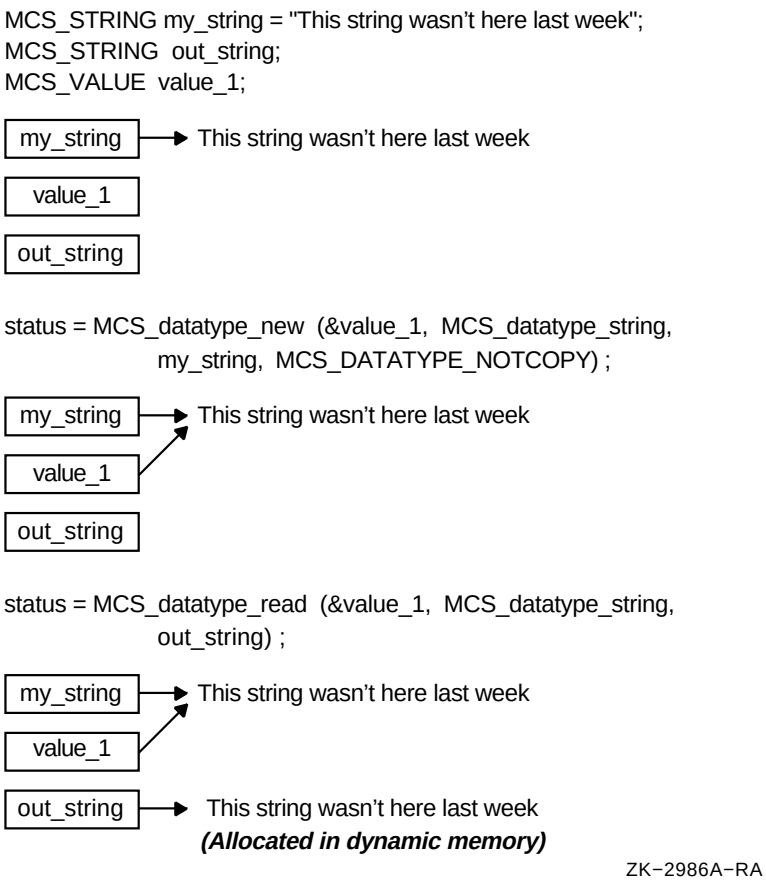
The following example retrieves the data that was stored in two value structures in the example in Section 3.2.1:

```
MCS_VALUE   value_int, value_string;
MCS_LONGINT x, string_length;
MCS_STRING  out_string;
MCS_STATUS  status;
   .
   .
   .
    status = MCS_datatype_read(
        &value_int,             /* The value structure */
        MCS_datatype_longint,   /* Name of type to read */
        &x);                    /* Value is placed in x */
   .
   .
   .
    status = MCS_datatype_length(
        &value_string,          /* Value structure containing string */
        &string_length,         /* Length of string returned here */
        MCS_datatype_string);   /* Type of data contained */
    out_string = malloc(string_length);

    status = MCS_datatype_read(
        &value_string,
        MCS_datatype_string,
        out_string);
   .
   .
   .
    free(out_string);
```

In this example, *MCS_datatype_length* obtains the length of the string before allocating memory to contain it. Section 3.2.4 describes *MCS_datatype_length* and other routines that manipulate value structures. Figure 3–2 shows the use of the *MCS_datatype_read* routine.

Figure 3–2 Reading Data from a Value Structure



The *MCS_datatype_read* routine also allows you to retrieve data in a different, but compatible, data type from that used by the value structure. For example, the string in Figure 3–2 could have been retrieved into a string descriptor instead of a null-terminated string. To do this, specify MCS_STRINGDSC in the call to *MCS_datatype_read* and supply an initialized dynamic string descriptor instead of a buffer. Similarly, you can read an integer into a floating-point form.

3.2.3 Freeing Data in Value Structures

The *MCS_datatype_free* routine frees memory associated with a value structure, provided that the *MCS_datatype_new* call that loaded the value structure specified MCS_DATATYPE_ISCOPY. If the call specified MCS_DATATYPE_NOTCOPY (the value structure refers to the value you supplied, not its copy of that value), the memory is *not* freed. In this situation, the application is responsible for managing this memory.

The *MCS_datatype_free* routine invalidates but does not deallocate the value structure. You can reuse the value structure.

MCS_datatype_free operates on value structures that store lists and scans as well as on all the other data types that you can specify with *MCS_datatype_new*. In the case of a list, *MCS_datatype_free* frees the memory associated with each value structure in the list, just as if you had called *MCS_datatype_free* on each list element. (See Section 3.10.1 for a complete description of this operation.)

3.2.4 Other Operations on Value Structures

The following routines manipulate value structures or return information about them:

- *MCS_datatype_copy*—Creates a copy of a value structure and the value that it contains.
- *MCS_datatype_datatype*—Returns the data type of the data stored in a value structure.
- *MCS_datatype_length*—Returns the length of the data stored in a value structure. Call this routine to find out how much space to allocate before calling *MCS_datatype_read*.
- *MCS_datatype_compare*—Compares the values stored in two value structures and indicates if they are equal or which is larger.

See the descriptions of these routines in Chapter 8.

3.3 Numeric Data Types

Oracle CDD/Repository defines the following numeric data types:

- MCS_SMALLINT—16-bit integer
- MCS_LONGINT—32-bit integer
- MCS_FLOAT—Single-precision floating-point number
- MCS_DOUBLE—Double-precision floating-point number

There are no special Oracle CDD/Repository callable interface routines to manipulate instances of these types.

3.4 BOOLEAN Data Type

Oracle CDD/Repository defines the MCS_BOOLEAN data type to represent true/false values. A variable of this type can take the following values:

Truth Value	C Symbolic Name OpenVMS Symbolic Name	Numeric Value
True	MCS_TRUE MCS$K_TRUE	1
False	MCS_FALSE MCS$K_FALSE	0

There are no special Oracle CDD/Repository callable interface routines to manipulate instances of these types.

3.5 String Data Types

Oracle CDD/Repository defines two string data types:

- MCS_STRING—Pointer to a null-terminated string used if your application is coded entirely in C
- MCS_STRINGDSC—OpenVMS string descriptor used if you call OpenVMS Run-Time Library routines or routines written in OpenVMS languages other than DEC C

3.5.1 Null-Terminated Strings

A null-terminated array of characters is the string representation expected by standard C language string-handling functions. If your application is coded entirely in C, MCS_STRING is the easiest string representation to use.

Oracle CDD/Repository defines MCS_STRING as a pointer to an array of characters. An input string argument to an Oracle CDD/Repository callable interface routine is declared as follows:

```
MCS_STRING argument
```

The Oracle CDD/Repository callable interface routine expects a pointer to a string that you have allocated and initialized. Some Oracle CDD/Repository callable interface routines return string values in an argument; these are declared as a pointer to a pointer to a string:

```
MCS_STRING *argument
```

For these arguments, the Oracle CDD/Repository callable interface routine allocates space for the string in memory, and sets *argument* to point to the string. The application is responsible for freeing this memory.

If you use *MCS_datatype_read* to retrieve a null-terminated string from a value structure, Oracle CDD/Repository expects a pointer to a buffer that is sufficiently long to hold the string. You can find the length of the string (including the null byte at the end) by using the *MCS_datatype_length* routine before calling *MCS_datatype_read*.

3.5.2 String Descriptors

OpenVMS languages other than C expect string arguments in the form of an OpenVMS string descriptor. If your application uses a language other than C, use MCS_STRINGDSC strings because they allow you to pass strings directly between the repository and the language.

A string descriptor is a structure that contains (among other information) a pointer to the string, the length of the string, and the type of the descriptor. There are two types of descriptors:

- dynamic (variable-length) descriptor types

  Use this type of descriptor for output arguments if you want Oracle CDD/Repository to allocate the memory for the full length of the output string, set the pointer in the descriptor to point to the allocated memory, and set the length in the descriptor to the length of the returned string.

- nondynamic (fixed-length) descriptor types

  Use this type of descriptor for output arguments if you want to allocate the buffer and set the pointer and length in the descriptor appropriately. Oracle CDD/Repository may truncate the returned string to fit in the buffer.

It is the responsibility of the application to initialize string descriptors appropriately. You can declare the character string in your implementation language or allocate an MCS_STRINGDSC variable and set its fields. The application also must call an appropriate routine, such as STR$FREE_DX, to deallocate a dynamic string that was allocated by Oracle CDD/Repository.

For more information on using OpenVMS string descriptors, see the OpenVMS documentation on system services.

3.6 Date and Time Data Types

Oracle CDD/Repository defines the MCS_VMSTIME time data type, which is a 64-bit datetime value based on the OpenVMS operating system.

3.7 Memory Block Data Type

Oracle CDD/Repository defines the MCS_MEMBLOCK data type to represent blocks of memory whose contents are meaningful to and managed by the application, not by Oracle CDD/Repository. Use this data type if your application stores data in a property (instead of in a file) but does not make that data directly available to Oracle CDD/Repository or to other applications. Your application can use GETPROP to retrieve the memory block from the property that stores it, examine and manipulate the data in memory, and then use SETPROP to place the memory block back in the repository as the value of the property.

The format of a memory block is one longword followed by a contiguous block of memory that contains the data. The longword contains the length of the data block in bytes.

3.8 Element ID Data Type

Oracle CDD/Repository defines the MCS_ELEMENTID data type to represent single elements in the repository. An element ID (or elmID) is an instance of MCS_ELEMENTID, and identifies a single element. Many Oracle CDD/Repository callable interface routines accept element IDs as input or output arguments, and the value of many properties is an element ID. If you send a message to an element, identify the element by its element ID.

The value of an element ID is valid only within a single session. For example, you cannot get an element ID that represents some element of interest, hold that value while you end one session and begin another, and then use it to refer to the same element. (*Persistent element identifiers* do identify the same element across sessions.)

You can think of an element ID as a pointer to an element, but it also carries information about the following:

- Current session—The *MCS_elmID_getSession* routine returns this information.
- Current persistent process—The *MCS_elmID_getPersistentProcess* routine returns this information.
- Current context—The *MCS_elmID_getContext* routine returns this information.

A number of functions provide information about elements given their element IDs or provide an element ID given other information about an element. These functions include the following:

- *MCS_element_getByName*—Returns the element ID of a named element.
- *MCS_element_getName*—Returns the name of a specified (by element ID) element.
- *MCS_element_getType*—Returns the element ID of the ELEMENT_TYPE element that represents the type of an element.
- *MCS_element_getSubTypeList*—Returns a list of the element IDs of the immediate subtypes of an element type.
- *MCS_element_getSuperTypeList*—Returns a list (with a single member) of the supertype of an element type.
- *MCS_elmID_copy*—Copies information from one element ID to another.
- *MCS_elmID_equal*—Determines if two element IDs refer to the same element.
- *MCS_elmID_isNull*—Determines if an element ID has a null value.
- *MCS_elmID_isSubtype*—Determines if one element type is a subtype of another.

See the description of each of these routines in Chapter 8 for more information.

The following routines convert between session-specific element IDs and persistent element identifiers:

- *MCS_elmID_export_persistent*—Creates a persistent element identifier from an element ID.
- *MCS_elmID_import_persistent*—Creates an element ID that you can use in the current session from a previously created persistent element identifier.

3.9 Scan Data Type

Scans are unordered lists of element IDs. (For some scan-valued properties, the first element of the scan has significance, but the order of the remainder of the scan is undefined.) Many element types have scan-valued properties.

There are two types of scans:

- scans that traverse relationships and return element IDs
- computed scans

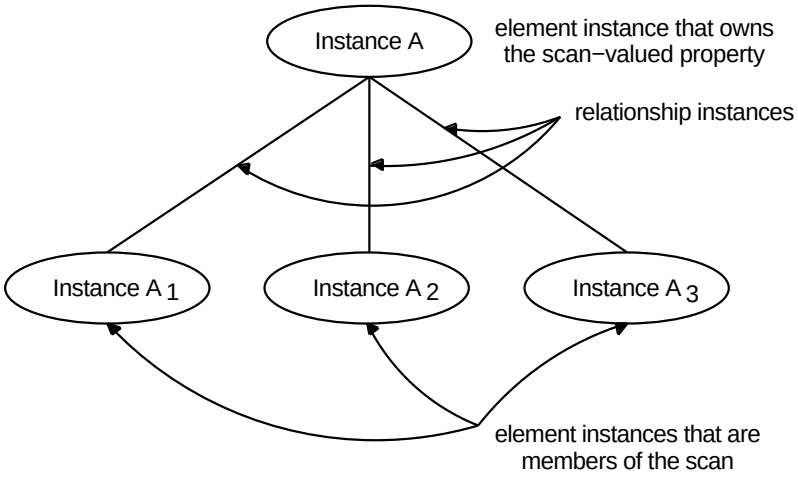
A scan may contain the members of a collection or the logical descendants of a version.

Unlike other types, such as numbers and strings, a scan cannot exist except as a value stored in a value structure. All routines that manipulate scans accept their scan arguments in the form of value structures.

The scan values themselves might be stored, or might be calculated at run time—your program does not need to know.

Scan routines that traverse relationships allow your program to access both the elements that belong to the scan and the relationships that connect them to the target element. Figure 3–3 shows a logical picture of a scan and the routines that access it.

Figure 3–3 Scans



To examine a scan property, use the following:

1. GETPROP to get the scan value
2. scan routines (see Section 3.9.1) to get each element ID of the scan in turn

To change a scan property, use the following:

1. GETPROP to obtain the value
2. scan routines (described in sections Section 3.9.2 through Section 3.9.5) to examine and modify the value
3. SETPROP to put the changed value in the property

3.9.1 Accessing Scan Contents

A number of routines allow you to traverse the element IDs in a scan or to locate a particular element ID in a scan.

A scan has a current position (one element ID in the scan is the current element ID). Operations that access scans set the current element ID and/or use it as a reference point. The operations are as follows:

- *MCS_scan_getNext*—Gets the element ID following the current one in the scan, and changes the current position to the retrieved element ID. For a fresh scan (one that has never been accessed) *MCS_scan_getNext* gets the first element ID in the scan. You can call *MCS_scan_getNext* to traverse the scan from beginning to end.

- *MCS_scan_getFirst*—Gets the first element ID in a scan, and sets the current position to that element ID.
- *MCS_scan_reset*—Resets a scan to the beginning.
- *MCS_scan_getCurrent*—Gets the current element ID of the scan. This is the element ID that was last retrieved by a call to one of the other MCS_ scan_get routines.
- *MCS_scan_getByName*—Returns the element ID for the named element. The search can start from the element ID following the current position or from the first element ID in a fresh or reset scan. You may use wildcard characters in the name.

The following example illustrates traversing a scan:

```
MCS_VALUE       scan_value;
MCS_ELEMENTID   elmID;
MCS_STATUS      status;
   .
   .
   .
(the scan is given a value by using GETPROP)
   .
   .
   .
    while (status != MCS_ENDFIND) {
      status = MCS_scan_getNext(
        &scan_value,        /* Value structure with scan */
        &elmID,             /* First element ID returned here */
        0);                 /* rID, ignored (see later) */

   .
   .
   .
(do some processing)
   .
   .
   .
        }
```

3.9.2 Adding an Element

To add new element identifiers to a scan-valued **relProp** property, your program should use the following steps:

1. Get an element ID for the element you want to add to the scan.
2. Send the **getProp** message to the element that owns the property to get the current value of the scan.

3. Call *MCS_arglist_findArg* to get the value structure for the scan.
4. Call *MCS_scan_insert* with the following:
   - the value structure that contains the present value of the scan
   - the element identifier of the item you want to add to the scan
5. Send the **setProp** message to write the changed value to the repository.

   This operation fails if the inserted element ID refers to the wrong type of element for the property.

Oracle CDD/Repository creates the appropriate relationships between the target element and the new element. The element is not stored until the **setProp** completes successfully.

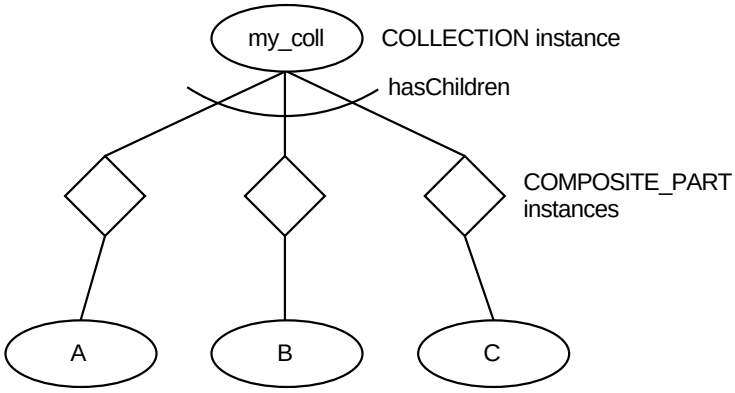
3.9.3 Accessing Relationships

Each element ID in a scan identifies an element that is connected to the target of the GETPROP message by a relationship. Relationships, like elements, are first-class objects; they can have properties. Sometimes you need to get or set a property value on a relationship. You can access a relationship through a scan.

Each of the MCS_scan_get routines that return an element ID (see Section 3.9.1) has an optional argument called *rID*, or relationship ID. If you supply a variable of type MCS_ELEMENTID in this argument, Oracle CDD/Repository fills it in with the ID of the relationship that connects the element returned in the routine. Figure 3–4 illustrates a simple collection and shows how *MCS_scan_getNext* can return both the element ID of a collection child (**A**) and the relationship ID of the COMPOSITE_PART instance that connects it to **my_coll**.

Figure 3–4 Getting Element and Relation IDs from Scans

```
MCS_VALUE  children_scan;
MCS_ELEMENTID  child,  rel;
MCS_STATUS  status;
(children_scan is given the value of the
hasChildren property on my_coll − not shown)
status = MCS_scan_getNext  (&children_scan, &child, &rel) ;
```



ZK−2988A−GE

You can use the relationship element ID with *MCS_scan_remove* or to manipulate the relationship instance, for example, by getting or setting its properties.

You also can set the properties of a relationship when you insert an element into a scan. To create a relationship that links the owner of the scan to the element being inserted, insert an element into a scan, then set the property value with SETPROP. If you use the *MCS_scan_insert_with_args* routine instead of *MCS_scan_insert*, you can specify properties to be set on the relationship. You must use this routine if the relationship is of a type that requires properties to be set at the time it is created. An example is HAS_RELATION_PROPERTY, which implements the **relPropDef** property on ELEMENT_TYPE.

To use the *MCS_scan_insert_with_args* routine to supply values that set the properties of the relationship, use the following steps:

1. Get an element ID, declare a value structure, and get the scan-valued property (by using **getProp**) as you would for *MCS_scan_insert* (see Section 3.9.2).
2. Declare a value structure and use it to create an argument list that contains the list of properties required by the relationship.
3. Call *MCS_scan_insert_with_args* and specify the following:
   - value structure containing the present value of the scan
   - element identifier of the item you want to add to the scan
   - argument list that contains the values for the relationship properties

   MCS_scan_insert_with_args copies the argument list, so you can free the list immediately.
4. Send the **setProp** message to write the changed value to the repository.

Oracle CDD/Repository adds the element to the scan by creating the relationship you specify with the property values you provide.

To remove an element from a scan, provide the relationship ID to *MCS_scan_remove*, not the element ID. The effect is the same, but it is more efficient to use the relationship ID.

3.9.4 Removing an Element

To remove an element from a scan-valued property, use the following steps:

1. Get an element ID for the element you want to take out of the scan, or a relation ID for the relationship you want to delete.
2. Declare a value structure to receive the present value of the scan.
3. Dispatch the **getProp** message to the element that owns the scan.
4. Call *MCS_scan_remove* with the following:
   - the value structure that contains the present scan
   - either the relation ID for the relationship to be removed (preferred method), or the element ID of the element to be removed from the scan, but not both
5. Dispatch the setProp message to the element to make the change permanent.

If the element ID you specify matches an element ID in the scan, Oracle CDD/Repository removes that element from the scan by deleting the relationship that connects it to the scan.

If you specify a relationship ID, Oracle CDD/Repository deletes the relationship directly, which removes the element from the scan.

The result is the same in either case. Deleting the relationship directly is more efficient, but supplying the element ID allows your program to do less processing directly.

3.9.5 Initializing a New Scan

You generally get a scan by using GETPROP to obtain the value of a scan-valued property. Even if you know the current value of the property is an empty scan, you must get its value with GETPROP, then insert elements in the scan.

You should create a new scan only if you are writing a computed property whose value is a scan. For example, if you dispatch **new** to a type that requires a scan property, you must supply that property value when you create the element. You cannot set the value after the element has been created.

In this situation, you can use the *MCS_scan_new* routine. This routine allocates and initializes an empty scan, and initializes a value structure to contain the scan. However, you must specify the element type that owns the property whose value you are creating (or supply the element ID of the element), and the name of the property. You cannot create a scan that is not meant to be the value of a specific property.

To create a scan to supply as the initial value for a property, use the following steps:

1. Declare a value structure.
2. Get an element identifier for the element type that defines the property you want to create.
3. Call *MCS_scan_new* to initialize the value structure to contain an empty scan. Your program must specify the following:
   - value structure
   - name of the property for which this scan is the value
   - element ID of the ELEMENT_TYPE element that owns the property in the type hierarchy

   Do not use any of the other optional arguments to *MCS_scan_new*.

4. Call *MCS_scan_insert* or *MCS_scan_insert_with_args* to put values into the scan.
5. Dispatch the **new** message with the scan in the argument list.

3.10 List Data Type

Oracle CDD/Repository defines the list data type and a number of operations to manipulate lists. A list is an ordered sequence, possibly empty, of value structures. Because a value structure can contain any kind of data, list entries can be of any data type, including a list.

Like scans, lists can exist only in value structures. All routines that manipulate lists accept their list arguments in the form of value structures.

The list routines allow you to do the following:

- create and delete lists
- retrieve list entries
- insert, remove, and set list entries
- find the size of lists

One important use of a list is to pass arguments to methods. If a list is used this way, it is called an argument list. Oracle CDD/Repository provides a special set of routines to manipulate argument lists (see Section 3.11). However, an argument list is only a specialized use of a list. It does not differ structurally from a list.

3.10.1 Creating and Deleting Lists

Use the *MCS_list_new* routine to create a new list. This routine allocates an empty list and initializes a value structure that you supply so that it points to the new list. You also specify the initial number of preallocated entries in the list and the number of entries to add each time the list needs to grow. (The list entries are preallocated to allow rapid insertion up to the number of preallocated entries. However, the current size of the list depends on how many entries have been added or removed. Lists are always created with a current size of zero.)

To create a list, your program should use the following steps:

1. Declare a value structure.
2. Pass this value structure to *MCS_list_new*; optionally, it can specify how many entries the list initially contains and how many entries should be added to the list each time it needs to grow.

Oracle CDD/Repository initializes the empty list by using the parameters you specify. The current size of the newly created list is zero.

The previous technique is the same for both argument lists and other kinds of lists.

The *MCS_list_free* routine frees a list and the memory referenced within that list if the memory was allocated by Oracle CDD/Repository. Calling *MCS_list_free* on a list is equivalent to calling *MCS_datatype_free* on each list element, then freeing the memory occupied by the value structures in the list. *MCS_datatype_free* deallocates the memory pointed to by the value structure if you initialized the value structure by using the MCS_DATATYPE_ISCOPY argument.

If a list entry is another list, *MCS_list_free* recursively calls *MCS_list_free* on that entry. This is particularly useful in the case of an argument list for GETPROP, SETPROP, or NEW, if the argument list contains a list of properties to set or get.

3.10.2 Retrieving a List Entry

You access list entries by their index in the list. The first entry in the list has index number zero (0); the last entry has the index MCS_LIST_END (MCS$K_LIST_END in the OpenVMS binding).

The *MCS_list_get* routine copies a specified list entry into a value structure that you provide. The *MCS_list_get* routine never copies the data pointed to by the list entry; therefore, following the operation, the list entry and the value structure copy of it point to the same data.

To read a particular entry, your program should use the following steps:

1. Define a value structure to receive the information.
2. Pass the value structure to *MCS_list_get*, specifying the index of the item you want.

Oracle CDD/Repository sets the value structure to point to the same data as the list entry. If you free the entry, you corrupt the list.

3.10.3 Inserting, Removing, and Setting List Entries

To insert an entry in a list, use the *MCS_list_insert* routine. Specify the index of the entry before which you want to insert the new entry. For example, to create a new first entry, specify index zero (0).

To add an entry to the list, your program should use the following steps:

1. Call *MCS_datatype_new* to package the data into a value structure. (If the value was returned by a method, this step is not necessary.)

2. Call *MCS_list_insert* with an initialized value structure and an index:
   - Zero (0) adds the entry to the front of the list. If the list already contains entries, the entries are moved out by one position.
   - An integer adds the new entry in front of the item identified by the integer. If the specified index already exists, the entries that follow are moved out by one position. If the integer is larger than the current size of the list, the error MCS_INDEXTOOLARGE is returned.
   - MCS_LIST_END adds the entry to the end of the list.

   The list is zero-based.

The current size of the list is increased by 1.

Oracle CDD/Repository copies the value structure but not the data.

To remove an entry from a list, use the *MCS_list_remove* routine, specifying the index of the entry to remove. This routine is equivalent to calling *MCS_datatype_free* on the value structure in that list position, then deallocating the value structure and removing it from the list.

To remove an entry from a list, your program should call *MCS_list_remove*, specifying the index of the entry to remove, as follows:

- An integer value removes the entry at the specified position.
- Zero (0) removes the first entry in the list.
- MCS_LIST_END removes the last entry in the list.

Oracle CDD/Repository performs the following actions:

- Frees the memory associated with the entry in the list position you specified. If the list entry is a list, the entire list is freed.
- Removes the entry from the list.
- Adjusts the current size and structure of the list.

Note

If two value structures point to the same data, and your program frees one of those structures, the remaining value structure points to bad memory.

The *MCS_list_set* routine changes a specified list entry. This routine is equivalent to removing the old list entry with *MCS_list_remove*, then inserting the new value structure in its place.

To use this routine, your program should use the following steps:

1. Declare a value structure that contains the new value.
2. Call *MCS_list_set* with the new value structure and the index of the entry to be replaced. MCS_LIST_END replaces the last entry in the list.

Oracle CDD/Repository removes the old entry and frees any memory associated with it, as if you had called *MCS_list_remove*, and then inserts the new entry in its place, as if you had called *MCS_list_insert*. There is no change to the list size.

3.10.4 Finding the List Length

The *MCS_list_getSize* routine returns the current number of entries in a list. If you create a list, you specify an initial size and increment for the list, and Oracle CDD/Repository allocates enough memory to contain the specified size of the list. The size returned by this routine, however, depends solely on how many entries have been inserted and removed since the list was created. You cannot access a list entry that lies outside the current size of the list as returned by *MCS_list_getSize*.

To find out how many entries are in the list at a given time, your program should use the following steps:

1. Declare a value of data type MCS_LONGINT to receive the count of items in the list.
2. Pass this value to *MCS_list_getSize* along with the value structure that contains the list whose length you wish to determine.

3.11 Argument Lists

Many of the methods invoked as the result of a message require additional information to perform the action you requested. You pass this additional information in a special list called an **argument list**.

The entries in the argument list specify individual arguments needed by the message you dispatch. Some of these arguments are required; others are optional. Each of these entries is called an **argument specifier**. Each argument specifier consists of the following:

- name of the argument
- value structure to contain the value of the argument
- status field

Argument specifiers represent both input and output parameters.

The entries in an argument list are referred to by name, not by position as in a regular list, so it does not matter in what order you specify them.

You may receive an "invalid datatype" error if you do either of the following:

- supply an argument list to an MCS routine that expects a regular list
- supply a regular list to a routine that expects an argument list

These actions also may cause an access violation because of the pointers in the lists.

If your program creates an argument list, it first creates an argument specifier that describes each argument. It then includes those argument specifiers in the argument list.

Oracle CDD/Repository provides the following routines to manipulate argument lists and argument specifiers:

- *MCS_arglist_addArg*—Adds arguments to an argument list.
- *MCS_arglist_findArg*—Reads (by name) an argument from an argument list.
- *MCS_arglist_getArg*–Reads (by index) an argument from an argument list.
- *MCS_arglist_setNameValue*—Sets (by name) the value of an argument.
- *MCS_arglist_setIndexValue*—Sets (by index) the value of an argument.

3.11.1 Building Argument Lists

To build an argument list, your program should use the following steps:

1. Call *MCS_list_new* to create the list, as for any other list.
2. Create the value structures as needed to contain the argument values.
3. Call the *MCS_arglist_addArg* routine once for each argument it adds to the argument list. For each call, your program supplies the following:
   - symbolic name of the argument
   - value structure that contains the value
   - status variable for the status field
   - argument list that receives the argument

Oracle CDD/Repository adds the argument specifier to the argument list. Because your program refers to the arguments by name, the order is unimportant.

3.11.2 Embedding Argument Lists

If you read or set property values, you specify the properties you want to read or write in an argument list. The argument list is then embedded in the main argument list. To build the second argument list, your program should use the following steps:

1. Create the list by calling *MCS_list_new*.
2. Create the value structures to hold the property values, as follows:
   - If your program is setting properties, it should provide the values for the value structures.
   - If your program is reading properties, it should supply 0 (zero) as the value parameter.
3. Call *MCS_arglist_addArg* once for each property whose value you wish to read or write.
4. To add this list of properties to the main argument list, call *MCS_arglist_addArg* with the following argument values:
   - The argument name is MCS_arg_arglist.
   - The value structure is the argument list you just created.

Oracle CDD/Repository reads or writes each property you specify and fills the status field of each entry with either a success message or a value that indicates the reason for failure.

It is possible for a read operation as a whole to succeed while some property operations fail. In this case, *MCS_dispatch_op* returns the status MCS_SOMEFAILED. The status field for each property indicates if that property was read and if not, why.

If any property cannot be written for any reason, the entire operation fails. *MCS_dispatch_op* returns with a status of MCS_SETPROPFAILED. The status field in the argument list indicates the property operation that failed and why. Your application should use the *MCS_session_transaction_term* routine to roll back any changes that were made. Specify MCS_TRANSACTION_ABORT as the value to the abort_flag argument. (Refer to Chapter 8 for additional information.)

3.11.3 Modifying List Entries

To modify entries in the argument list by index or by name, call *MCS_arglist_setNameValue* to set the value and status fields of the argument you name. You also can call *MCS_arglist_setIndexValue* to set the value and status fields of an argument you specify by position. Because the index is zero-based, zero (0) indicates the first entry.

Oracle CDD/Repository finds the argument specifier you want to modify, frees its old value and status, and sets the value to the passed-in value.

3.12 Returned Arguments

Your program can read the returned arguments either by position, by index, or by name, without regard to position.

3.12.1 Reading by Name

To read output arguments when you know the name of the argument you want to read, your program calls *MCS_arglist_findArg* with the following parameters:

- argument list that contains the argument
- name of the argument
- value structure that receives the returned value
- status variable to receive the status field of the argument specifier
- integer passed by reference to receive the index of the item (indicates the position of the argument in the list)

If the argument name is not in the list, the error MCS_ARGNOTFOUND is returned.

3.12.2 Reading by Index

To read output arguments when you know the position of the argument in the list, your program should use the following steps:

1. Initialize a string to receive the name of the argument.
2. Call *MCS_arglist_getArg* with the following parameters:
   - argument list that contains the argument
   - variable of type MCS_LONGINT that specifies the position of the argument

- value structure to receive the returned value
- status variable to receive the status field of the argument specifier
- string to receive the name of the argument

4
Working with Elements

There are two broad categories of operations you can perform on elements:

- Working with instances of existing elements, which includes the following tasks:
  - creating new instances of a type
  - reserving element instances that have already been created
  - replacing controlled versioned objects
  - canceling a reservation
- Creating new elements, which includes the following tasks:
  - creating element types
  - changing types
  - preventing incompatible changes

This chapter describes the tasks involved with each set of operations.

4.1 Creating Instances

To create a new instance of a type, your program should use the following steps:

1. Create an argument list that contains the arguments the **new** message needs, including:
   - argument list that contains any property values the new element requires (properties defined by the metadata as required)
   - argument to receive the element ID of the newly created type

   Section 3.11 describes how to create an argument list. (Refer to the *Oracle CDD/Repository Information Model Volume I* and *Oracle CDD/Repository Information Model Volume II* manuals for details about which properties an element of the type you are creating needs.)

2. Call *MCS_dispatch_op* with the following parameters:
   - element ID of the element type that you want to instantiate
   - **new** message
   - argument list

   Oracle CDD/Repository invokes the method to create an instance of that element type.

If versions of the element are permitted and the element is created while a context is open, the element is created under version control by default.

4.1.1 Placing an Element Under Control

Elements are created as uncontrolled objects by default if they are created without an open context. Elements can be created without a context, either through the Oracle CDD/Repository CDO interface without using a SET CONTEXT command or through the Oracle CDD/Repository callable interface if no context was open. To place such objects under version control, send the **control** message using the following procedure:

1. Find the element ID of the element you want to place under version control.

   Note

   You cannot place an element under control unless all members of all owned relationships are under control.

2. If you need to specify either of the optional arguments of the **control** message, create an argument list that includes the following:
   - A constant value indicating how much of the composite containing the element should be controlled. By default, only the element you specify is placed under control. You also can specify other related elements be placed under control (by using the values shown in Table 4–1): all elements in the composite, the specified element and all items that depend on it, or the specified element and all items on which it depends.
   - A list of elements that were controlled.

   Section 3.11 tells how to create an argument list.
3. Call *MCS_dispatch_op* with the following parameters:
   - element ID of the element you want under version control

- **control** message
- argument list

In response to this procedure, Oracle CDD/Repository places the element under version control. If the element was already under version control, the ISCONTROLLED error is returned.

Table 4–1 Option Values

Value	Meaning
MCS_TO_NONE	The operation applies only to this element. This is the default.
MCS_TO_TOP	The operation applies to this element and to all its direct and indirect ancestors to the top of the composite hierarchy.
MCS_TO_BOTTOM	The operation applies to this element and to all its direct and indirect descendants in the composite hierarchy.
MCS_TO_ALL	The operation applies to this element, to all its direct and indirect descendants in the composite hierarchy, and to all its direct and indirect ancestors to the top of the composite hierarchy.

4.1.2 Reserving an Element

To reserve a versioned element to make changes, your program should send the **reserve** message by using the following procedure:

1. Open a context and set top.
2. Get an element ID for the element you want to reserve.
3. Create an argument list to contain the arguments needed by **reserve**:
   - An argument to receive the element ID of the newly created ghost version. This argument is required.
   - The name of the branch to create if the element has successors in this line of descent. If you specify the branch name, the element is reserved on the branch you specify.
   - A constant value that indicates how much of the composite that contains the element should be reserved. By default, only the element you specify is reserved. You also can reserve other related elements (by using the values shown in Table 4–1): all elements in the composite, the specified element and all items that depend on it, or the specified element and all items on which it depends.

The argument is ignored if it cannot be applied.

- A list of elements that were reserved.

4. Call *MCS_dispatch_op* with the following parameters:
   - element ID of the element you wish to reserve
   - **reserve** message
   - argument list

Oracle CDD/Repository reserves the element by creating a new version of the element and placing it in the ghost state. Only the context that reserved the element can modify this reserved version. Any other context accessing the version sees the previous version. If no context is set, all applications can see the object, but no application can modify the object.

4.1.3 Replacing a Reserved Element

To make your changes permanent and make the element shareable, your program sends the **replace** message by using the following procedure:

1. Find the element ID of the element you want to reserve.
2. If you want to specify one or more of the optional arguments of the **replace** message, create an argument list that includes the following:
   - A constant value that indicates how much of the composite that contains the element should be replaced. By default, only the element you specify is replaced. You also can replace other related elements (by using the values shown in Table 4–1): all elements in the composite, the specified element and all items that depend on it, or the specified element and all items on which it depends.
   - A list of elements that were replaced.

   Section 3.11 tells how to create an argument list.
3. Call *MCS_dispatch_op* with the following parameters:
   - element ID of the element you want to replace
   - **replace** message
   - argument list

Oracle CDD/Repository makes the element you specify shareable and immutable.

4.1.4 Canceling a Reservation

If you decide not to use the changes you have made, your program can cancel the reservation. It uses the following procedure to send the **unreserve** message:

1. Find the element ID of the reserved element.
2. Create an argument list to contain the arguments the **unreserve** message needs:
   - The element ID of the predecessor of the unreserved element.
   - A constant value indicating how much of the composite containing the element should be unreserved. By default, only the element you specify is unreserved. You also can unreserve other related elements (by using the values shown in Table 4–1): all elements in the composite, the specified element and all items that depend on it, or the specified element and all items on which it depends.
   - A list of elements that were unreserved.

   Section 3.11 tells how to create an argument list.
3. Call *MCS_dispatch_op* with the following parameters:
   - element ID of the reserved element
   - **unreserve** message
   - argument list

Oracle CDD/Repository cancels the reservation by deleting the ghost element and any associated files.

If the element for which you are canceling the reservation is the top of the collection hierarchy, the entire file directory structure corresponding to the hierarchy is deleted. The predecessor of the element in this line of descent becomes the new top, and the repository system creates a new file directory structure to correspond to the new collection hierarchy.

Note

You cannot unreserve metadata elements.

4.2 Creating Element Types

If your application needs to extend the element type hierarchy by creating a new element type, you generally need to add a new leaf node to the element type hierarchy tree by specifying that the new element type is a subtype of an existing type. The new type represents objects that are a more specialized variety of the objects represented by the supertype.

For example, a programming language that integrates with the repository can create an element type to represent its source files. Because source files are text files, the language specifies that the supertype of its new type is TEXT. Elements of the new type automatically inherit the properties and methods possessed by TEXT elements. What differentiates a new type from its supertype are the new properties and methods that it defines, refines, or disallows.

Before you create a new type, examine the properties and relationships of the object the new type is to model. Compare those properties to the characteristics of existing element types to identify which type most closely models those properties. This type is the supertype of the new type.

If the object that you want to model is more specialized than the object modeled by the supertype, you need to create new properties and relation types, refine inherited methods, and define new methods.

Note

You cannot create metadata elements on a branch.

4.2.1 Reserving the Metadata Collection

Types and other repository metadata objects not only describe themselves, they also describe their structure of instances, including elementTypes, relationTypes, and propertyTypes.

The metadata for a particular repository is part of the CDD$METADATA collection. This includes the contents of the type hierarchy, and any properties, data types, methods, messages, message arguments, and validations that objects in the type hierarchy use. Those objects that are part of the system metadata make up the schema of instances that may be stored in a particular repository.

The CDD$METADATA collection is a controlled object in the CDD$METADATA_PARTITION partition, which is a partition referenced by every partition hierarchy you create.

The active schema collection (metadata) describes the repository schema from which new objects are created.

Because adding a new ELEMENT_TYPE element to the type hierarchy changes the repository schema, you must reserve the CDD$METADATA collection. The new types you create are not instantiable until you replace the metadata collection.

If you create a subtype of a metadata object, do not attempt to use the new subtype as a metadata object. For example, you cannot dispatch to a validation even though the validation is a subtype of method.

Note

You cannot unreserve metadata elements.

4.2.2 Defining Element Types

To define a new type, use the following steps:

1. Reserve the CDD$METADATA collection.
2. Invoke the NEW method, specifying ELEMENT_TYPE, RELATION_TYPE, or PROPERTY_TYPE. You cannot create new DATA_TYPE element types.

   In response, the repository performs the following functions:

   - Creates an instance of the type you specified and sets any default or system properties.
   - Sets the protocol for the new type version number to 0.0. This new elementType definition is not instantiable until it and the CDD$METADATA collection are replaced into the CDD$METADATA_ PARTITION partition. After a type becomes part of the schema, it is assigned a major and minor version number.
   - Generates a unique tag value if MCS_tag is not specified. MCS_tag associates type definitions across physical repositories. The first word should contain the product facility code and the second word contains a unique number for the type.
   - Assigns the new type a name in the CDD$PROTOCOLS directory.
3. Supply an argument list to set any additional properties, such as:
   - methods
   - propDef

- legalOwners
- legalMembers
- superTypes

Before you set the **legalOwners** property, be sure the owner is reserved.

Caution

You must thoroughly test your changes in a private test repository before you check them into the system schema. Because you cannot reverse changes to the schema, you may change the repository in a way that permanently corrupts the repository database. (See Section 4.4 to find out what changes are incompatible.)

4.2.3 Replacing the Metadata Collection

Replacing the CDD$METADATA collection triggers a repository rebuild, which locks the entire physical repository.

The repository performs the following steps:

1. Computes inheritance.
2. Creates storage for instances of the new type.
3. After the collection is replaced, sets any new types it contains with protocol version 0.0 to 1.0.

If the element types or relation types that are part of the collection being replaced do not have a supertype, you receive the error CDD$_SUPREQ.

4.3 Changing Element Types

You can modify any reserved type. To modify meta-metadata types supplied by Oracle CDD/Repository, you must have the privilege CDD$EXTENDER. You must modify the ACL for the type before you attempt to modify the type. If you have system privilege, you can change other types supplied by Oracle CDD/Repository.

The following is the recommended sequence of steps to modify the system metadata:

1. Reserve the CDD$METADATA collection.
2. Reserve the types you want to modify.
3. Modify the types as needed.

4. Replace to bottom the CDD$METADATA collection.

If you modified an existing type, the existing instances are still bound to the old types (this is why you cannot purge types.) New instances, however, are bound to the new types.

New instances of a type are always defined by using types in the active schema. In addition, instances defined under previous versions of that type are automatically upgraded to the latest version of that type.

4.3.1 Using the RESERVE Method

If you reserve a type, method, or message, you must set the value of the top property to be the CDD$METADATA collection. RESERVE sets the closure argument to MCS_TO_TOP. The closure argument specified as MCS_TO_ BOTTOM generates an error. The schema must be reserved to TOP because a legitimate schema allows only one active version of any type to be used by members of the schema.

4.3.2 Using the REPLACE Method

If you replace the CDD$METADATA collection, set the closure argument to MCS_TO_BOTTOM. If you do not, the REPLACE method sets the value of closure to MCS_TO_BOTH. The repository performs the following steps:

1. Performs inheritance on the changed type.
2. Updates the protocol version number for all changed types. An upwardly compatible change increments the minor version of the type number by 1 but does not change the major version number. An incompatible change increments the major version by 1 and sets the minor version to zero. See Section 4.4 for a description of compatible and incompatible changes.

4.4 Compatible and Incompatible Changes

A compatible change implies an extension to the existing definition. An incompatible change implies that the change is more restrictive. Existing definitions may lose data. Additionally, applications written against the types may break.

For example, if you make an upwardly incompatible change to a type that Oracle CDD/Repository uses, Oracle CDD/Repository may fail. In addition, your repository can become corrupted beyond repair.

4.4.1 Element Type and Relation Type Changes

Upwardly compatible changes to elementTypes or relationTypes include the following:

- adding optional properties
- adding legal owners or members
- adding a method
- making a required property optional
- changing all mutable properties, including the following:
  - ACL of the elementType or relationType
  - ACLs for instances of the elementType or relationType

Incompatible changes to elementTypes or relationTypes include the following:

- adding a required property
- making an optional property required
- removing a legal owner or member
- removing a property
- using a new version of a property type that has been incompatibly changed

If a type inherits an incompatible change, that type is incompatibly changed.

4.4.2 Property Type Changes

Compatible changes to properties include the following:

- changing the data type to a less restrictive data type (changing an integer type to a larger integer type, a text type to a longer text type, or a numeric type to a text type)
- changing mutable properties, including the ACL of the property type
- changing the property validation to be more restrictive
- adding a history entry

Incompatible changes to properties include the following:

- changing the data type to a more restrictive data type
- changing the scale factor

All other changes are considered to be an incompatible modification of the property.

5
Working with Methods

Applications perform operations on Oracle CDD/Repository elements by sending messages to them. The message requests a general action. The element responds to the message by invoking a method that implements the action in a way that is appropriate for elements of that type.

This chapter includes the following topics:

- message dispatching
- using *MCS_dispatch_op*
- refining methods
- validating refined methods
- refining external code methods
- refining external program code methods

5.1 Dispatching Operation

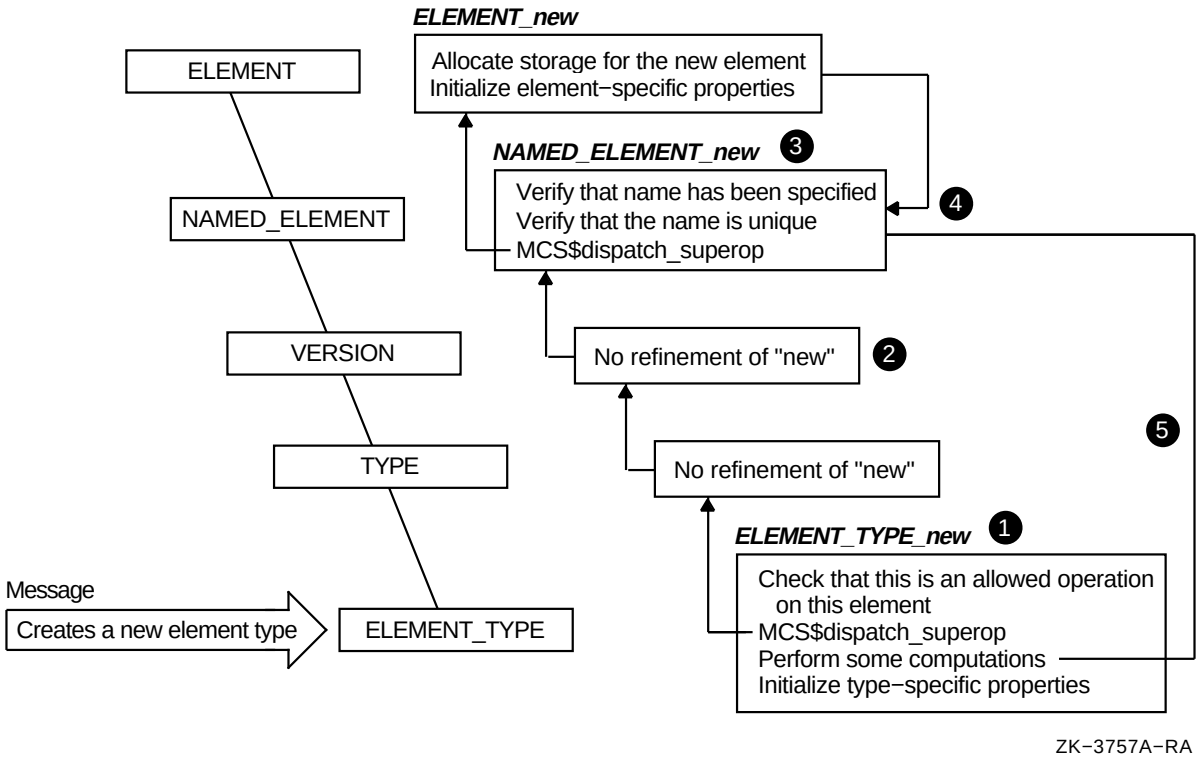
If you send a message to an element, the method dispatcher looks for a method defined by that element type to respond to the message you sent. If it finds a method, it invokes it. If not, the method dispatcher looks for a method defined by the supertype of the type. The method dispatcher works its way up the type hierarchy until it either finds a method to invoke or reaches the root of the hierarchy, in which case it returns an error indicating that the message is not recognized.

A method does only the processing that is specific to the type that defines it. For example, methods defined by BINARY (which represents files) only do processing that manipulates files. They do not do processing that manipulates elements within the repository; instead, they use the methods provided by BINARY supertypes (notably VERSION). In this way, they inherit behavior from the supertype, and do not have to duplicate the supertype's code.

A method that inherits behavior needs a way of indicating where in its processing it wants that behavior to happen. The *MCS_dispatch_superOp* routine is provided for this purpose. A method calls this routine to invoke the method that the supertype provides for a message.

Figure 5–1 illustrates method dispatching. The message NEW is sent to the ELEMENT_TYPE element named **ELEMENT_TYPE**, with the intention of creating a new element type.

Figure 5–1 Method Dispatching



Key to Figure 5–1:

❶ The method dispatcher locates and invokes the ELEMENT_TYPE_new method. This method performs some operations that are specific to the type, then calls *MCS_dispatch_superOp*.

❷ The dispatcher then looks for the nearest supertype that defines a "new" method. The TYPE and VERSION element types shown in this example do not define "new" methods, so the method dispatcher continues its search up the hierarchy until it finds one defined for the NAMED_ELEMENT type.

❸ The NAMED_ELEMENT_new method contains a call to *MCS_dispatch_superOp*. Therefore, the dispatcher looks for the nearest supertype that defines a "new" method. It finds one defined for ELEMENT and invokes the method.

❹ ELEMENT is at the top of the hierarchy. Therefore, when ELEMENT_new completes, it returns control to NAMED_ELEMENT_new.

❺ NAMED_ELEMENT_new completes and returns control to ELEMENT_TYPE_new, which performs some operations that are specific to the type and exits.

Note

If ELEMENT_TYPE_new did not exist, processing of the NEW message would have been handled by the NAMED_ELEMENT_new and ELEMENT_new methods.

5.2 Invoking MCS_dispatch_op

MCS_dispatch_op has three parameters, two of them required.

The first argument specifies the element to which the message is sent. Its value is the element ID of the element.

Oracle CDD/Repository assigns an element ID as part of several operations:

- If you know the element name, you can request an element ID by using the *MCS_element_getByName* routine.
- If you have an element ID, you can get an ID for the ELEMENT_TYPE element that represents its type by using the *MCS_element_getType* routine.
- Many properties have element IDs or scans of element IDs as their values. You can send the **getProp** message to the element that owns the property to return the element ID stored there.

The second argument to *MCS_dispatch_op* specifies the message being sent. Its value is a string that is the name of the message—usually one of the global variables defined for your language and binding (see Chapter 1). If you are sending a message that your application or another application has defined, supply a string containing the name of the MESSAGE element that defines the message.

The third argument to *MCS_dispatch_op* is an argument list that specifies arguments for the message.

Each message defines arguments that an application sends along with the message. Many messages have no arguments. Others require one or more arguments, and may accept optional arguments. The message descriptions in the *Oracle CDD/Repository Information Model Volume I* manual list the required and optional arguments that each expects.

Section 3.11 tells how to build and use argument lists.

Note

Be sure to clear your error stack between dispatches.

5.3 Refining Methods

Method refinement can take the following forms:

- A method can, during its processing, invoke the method for the supertype. This allows the method to take advantage of the behavior that the supertype defines for a message, leaving only the type-specific processing to the method.
- A method can invoke other methods that are not the methods of its supertypes.
- A method can perform its processing without invoking the method for the supertype. In this situation, the type does not inherit the supertype behavior for that message, but instead replaces it. The same technique is used if a type needs to respond to a message that its supertype does not recognize.
- A method can disallow a message for an element type. Setting the **funcType** property of the METHOD element to "MCS_METHOD_ILLEGAL" accomplishes this.

- A method can invoke the method of the supertype without performing any processing of its own. Setting **funcType** to "MCS_METHOD_SUPEROP" accomplishes this. This function method serves as a placeholder for a preamble, a postamble, or both. **Preambles** and **postambles** are sets of methods that execute before and after the method to which they are attached. (See the *Oracle CDD/Repository Architecture Manual* for more information on preambles and postambles.)

Regardless of the method refinement used, the process of associating a message, method, and element type is the same.

1. Reserve the CDD$METADATA collection.
2. Create the ELEMENT_TYPE element that represents the new type, if no existing type meets your needs.
3. Create the METHOD element that represents the implementation of a message for that type. You must supply several properties if you create instances of this element type.
4. Locate the MESSAGE element that represents the message. You can use *MCS_element_getByName* to get the element ID of the message. If you define a new message that this method implements, you need to create a MESSAGE element.
5. Set the value of the **implementsMessage** property of the METHOD element to the element ID of the MESSAGE element.
6. Insert the element ID of the METHOD element into the **methods** property on the ELEMENT_TYPE element.

Repeat this procedure for each method that the new type defines. This establishes the linkage that the method dispatcher follows when it selects a method to invoke.

If your application defines a new element type and associated methods, you must specify the action for each method. You first select the function type of the method. This in turn determines how you provide the action for that method.

5.4 Validating Refined Methods

If you refine a method, you may want to write a validation for that method. **Validations** allow you to check repository objects either before or (more typically) after you execute a method. For example, if your system does not allow any objects that start with the letter P to be created, you can write a validation to be executed after a NEW or SETPROP that reads the new object name and returns an error if it finds a name that starts with the letter P.

If you refine a method associated with one of the following Oracle CDD/Repository messages, the corresponding Version 4.n operations cannot define or change instances of that type unless you validate those operations:

- **NEW**
- **SETPROP**
- **RESERVE**
- **REPLACE**

The functions provided by **NEW** and **SETPROP** are performed explicitly through the Version 4.n operations CDD$DEFINE_ELEMENT and CDD$CHANGE_ELEMENT (respectively). **RESERVE** and **REPLACE** functions are handled implicitly on Version 4.n change operations. You also must validate types that have refined methods to allow access to those types from the following:

- CDO commands DEFINE GENERIC and CHANGE GENERIC
- layered products integrated with Version 4.n

To validate a Version 4.n operation, create a validation object by using the same procedure you do for creating a method. The following list outlines the steps in the procedure:

1. Dispatch **NEW** to type VALIDATION.
2. Attach the validation to the appropriate type (type on which the method was refined).
3. Write the validation code (can be done before or after steps 1 and 2).

The VALIDATION type is a subtype of the METHOD type. Validations are similar to methods in that they are inherited and they can have executable code associated with them. The validation code should contain the special checks specified in the method refinement.

When Version 4.n operations or CDO commands are performed, Oracle CDD/Repository executes the validations you defined for the related operation. If the validation fails, the operation fails.

If you use only types that have refined methods through the Oracle CDD/Repository callable interface, you do not need validations. You can write validations for Oracle CDD/Repository callable interface operations, but the use of preambles and postambles is preferred.

Properties defined on type VALIDATION include all properties inherited from METHOD, as well as the properties described in the following subsections.

MCS_validationQuery

MCS_validationQuery provides compatibility with Version 4.n. For this property, the validation is defined through a query buffer, not through a validation routine.

MCS_validationApply

MCS_validationApply specifies operations to which validation applies. The data type is MCS_SMALLINT and the access is read/write. The following list contains the possible values:

- literal cdd$k_val_new = 1—Apply if you create an object.
- literal cdd$k_val_setprop = 2—Apply if you modify an object.
- literal cdd$k_val_free = 4—Apply if you delete an object.
- literal cdd$k_val_reserve = 8—Apply if you reserve an object.
- literal cdd$k_val_replace = 16—Apply if you replace an object.
- literal cdd$k_val_new_prot = 32—Apply if you create a new type.
- literal cdd$k_val_setprop_prot = 64—Apply if you modify a type.

Use cdd$k_val_new_prot or cdd$k_val_setprop_prot if you refine methods on a subtype of type TYPE.

Because the value for **MCS_validationApply** is interpreted as a bitmask, you can specify that a validation applies to multiple operations. For example, a value of cdd$k_val_new AND cdd$k_val_setprop indicates that the validation applies to both new and change operations.

MCS_validationAction

MCS_validationAction specifies action taken if the validation fails. The data type is MCS_SMALLINT and the access is read/write. Its values include:

- literal cdd$k_warn = 0—If validation fails, a message is placed in the error stack but the operation continues.
- literal cdd$k_fail = 2—If validation fails, the operation fails.

MCS_validationWhen

MCS_validationWhen specifies when the validations are executed. The data type is MCS_SMALLINT and the access is read/write. The following list contains the possible values:

- literal cdd$k_val_start = 1—Run validation before operation.
- literal cdd$k_val_end = 2—Run validation after operation.

- literal cdd$k_val_ci = 4—Run validation only if call is from the Version 4.n callable interface.
- literal cdd$k_val_mcs = 8—Run validation only if call is from the Oracle CDD/Repository callable interface.
- literal cdd$k_val_ci_mcs = 12—Run validation if call is from either the Version 4.n callable interface or the Oracle CDD/Repository callable interface.

Because the value is interpreted as a bitmask, you can specify values such as cdd$k_val_start AND cdd$k_val_ci. The resulting value causes the validation to be run before an operation is invoked from the Version 4.n callable interface.

MCS_associatedValidations

MCS_associatedValidations is a scan of all validations owned by the type. It is defined by the ELEMENT_TYPE type.

5.5 Method Categories

There are three broad categories of methods:

- External code methods, which are entry points associated with a METHOD element. They are called by Oracle CDD/Repository as subroutines, and execute in the address space of the caller. External code methods can invoke the method of the supertype during their processing. (See Section 5.6 for information on using external code methods.)
- External program methods, which are complete programs associated with a METHOD element. They are run by Oracle CDD/Repository rather than called, and execute in their own address space. They cannot dispatch the method MCS_DISPATCH_SUPEROP. (See Section 5.7 for information on using external program methods.)
- All other method types, including null, illegal, and superop. These methods require no action on your part other than setting the value of the **funcType** property on the METHOD element appropriately. (See the *Oracle CDD/Repository Architecture Manual* for more information on these method types.)

In addition to these method types, an element type may specify no action at all for a given message; that is, not create a METHOD element to respond to that message. In this situation, the new element type inherits both the message and the method of its supertype.

You also can attach preambles and postambles to methods as needed. (See the *Oracle CDD/Repository Architecture Manual* for more information on preambles and postambles.)

5.6 Refining External Code Methods

External code methods are called as entry points when their associated method is invoked. An external code method executes in the Oracle CDD/Repository address space. Package external code methods in shareable images. Control the packaging through the value of **funcType** on the METHOD element.

5.6.1 Method Function Calling Sequence

All method functions are called with two arguments:

- element ID of the element to which the message was sent
- argument list (called the dispatch list) that was sent with the message

The dispatch list contains named input and output arguments associated with the message. The method function uses the various argument-list manipulation routines to read and modify these arguments before returning. (See the *Oracle CDD/Repository Information Model Volume I* manual for information on messages and Chapter 8 of this manual for information on argument list routines.)

Some messages take no arguments; therefore, the dispatch list is considered an optional argument, and the caller of the method function may omit it by passing a zero by value. However, if the dispatch list is omitted when you send a message that has a required argument, an error results.

A method function is associated with a METHOD element, which in turn is associated with one element type–message pair. Because the method function knows which message it implements and for which type, the function does not require arguments that specify the message or the type.

A method function must return a value of type MCS_STATUS, that is, a longword condition value. If the function executes normally, this value is MCS_SUCCESS.

5.6.2 Method Functions and Transactions

Because functions that implement external code methods can be called only if a transaction is already active, these functions should never start transactions. This means that repository changes made by the method are undone if you roll back the transaction. In addition, the method can lock the repository against its caller if the transaction is read/write.

A method function that needs to manipulate files that are part of the file system controlled by Oracle CDD/Repository should use the Oracle CDD/Repository file operation routines, not the native file system facilities. The file operation routines provide two benefits:

- If the transaction that invoked the method aborts, the file operations are undone, which keeps your files synchronized with the repository.
- The file operation routines are portable between operating systems; native file system operations are not.

(See Chapter 8 for descriptions of the file operation routines.)

5.6.3 Invoking the Supertype Method

A method function that refines the response to a message invokes the supertype method by using the *MCS_dispatch_superOp* routine.

MCS_dispatch_superOp executes the method the supertype (or possibly one of its supertypes) has defined to respond to the message. Using this process, a method function only has to perform processing that is specific to the type that defines it.

Before you write a method that uses *MCS_dispatch_superOp*, you should understand how the process works. This allows you to write your method so that its processing is properly coordinated with that of *MCS_dispatch_superOp*. To understand the methods supplied with Oracle CDD/Repository, see the description of the corresponding message in the *Oracle CDD/Repository Information Model Volume I* and *Oracle CDD/Repository Information Model Volume II* manuals. Each message description contains descriptions of the methods that implement the message.

The *MCS_dispatch_superOp* routine is identical to the *MCS_dispatch_op* routine, except for an additional argument. This argument, the first in the calling sequence, identifies the type whose supertype method is to be invoked. More precisely, it is the element ID of the ELEMENT_TYPE element that defines the type on which this method is refined. Before you can call *MCS_dispatch_superOp*, you must obtain this element ID. Example 5–1 shows how to obtain the type element ID and call *MCS_dispatch_superOp*. The example contains a method function for X_TYPE that does nothing but invoke *MCS_dispatch_superOp* for the **close** message. (The example is for purposes of illustration only. Typically, you would not write a method function that only invoked *MCS_dispatch_superOp* and exited. Instead, you should use *MCS_dispatch_op* with a method function type of "MCS_METHOD_SUPEROP".)

Example 5–1 Calling *MCS_dispatch_superOp*

```
MCS_STATUS CLOSE_X_TYPE (
    MCS_ELEMENTID   *instance,
    MCS_VALUE       *dispatchList
    )
{
MCS_ELEMENTID       type_elmID,  /* element ID of any ELEMENT_TYPE element */
                    elementtype_elmID, /*  element ID of ELEMENT_TYPE type */
                    XType_elmID; /*  element ID of X_TYPE type */

    if ( ( status = MCS_element_getType ( ❶
        instance,
        &type_elmID ) ) != MCS_SUCCESS ) {
            return ( status );
        }
    if ( ( status = MCS_element_getType ( ❷
        &type_elmID,
        &elementtype_elmID ) ) != MCS_SUCCESS ) {
            return ( status );
        }
    if ( ( status = MCS_element_getByName ( ❸
        &elementtype_elmID,
        "X_TYPE",
        &XType_elmID,
        0 ) ) != MCS_SUCCESS ) {
            return ( status );
        }

    return (MCS_dispatch_superOp ( ❹
        &XType_elmID,
        instance,
        MCS_message_close,
        dispatchList ) );
}
```

Key to Example 5–1:

❶ The first call to element_getType gets the element ID of the ELEMENT_TYPE element that represents *instance* type. The goal of this call is to get the element ID of *any* ELEMENT_TYPE element.

❷ The second call to element_getType gets the element ID of the ELEMENT_TYPE element that represents the ELEMENT_TYPE. This element ID is then used as an argument to element_getByName.

❸ This call gets the element ID of the ELEMENT_TYPE element that represents X_TYPE.

❹ The call to *MCS_dispatch_superOp* uses the element ID obtained in the preceding call to indicate that the CLOSE method for the supertype of the X_TYPE is to be executed. The return value of the *MCS_dispatch_superOp* call becomes the return value of CLOSE_X_TYPE.

Some of the code in Example 5–1 might seem unnecessary; for example, use element_getType to find the type of the *instance*. You cannot use that technique because *instance* cannot be of type X_TYPE. CLOSE_X_TYPE may have been invoked because a subtype of X_TYPE was sent the **close** message. In that situation, the type of *instance* would be that of the subtype, not X_TYPE.

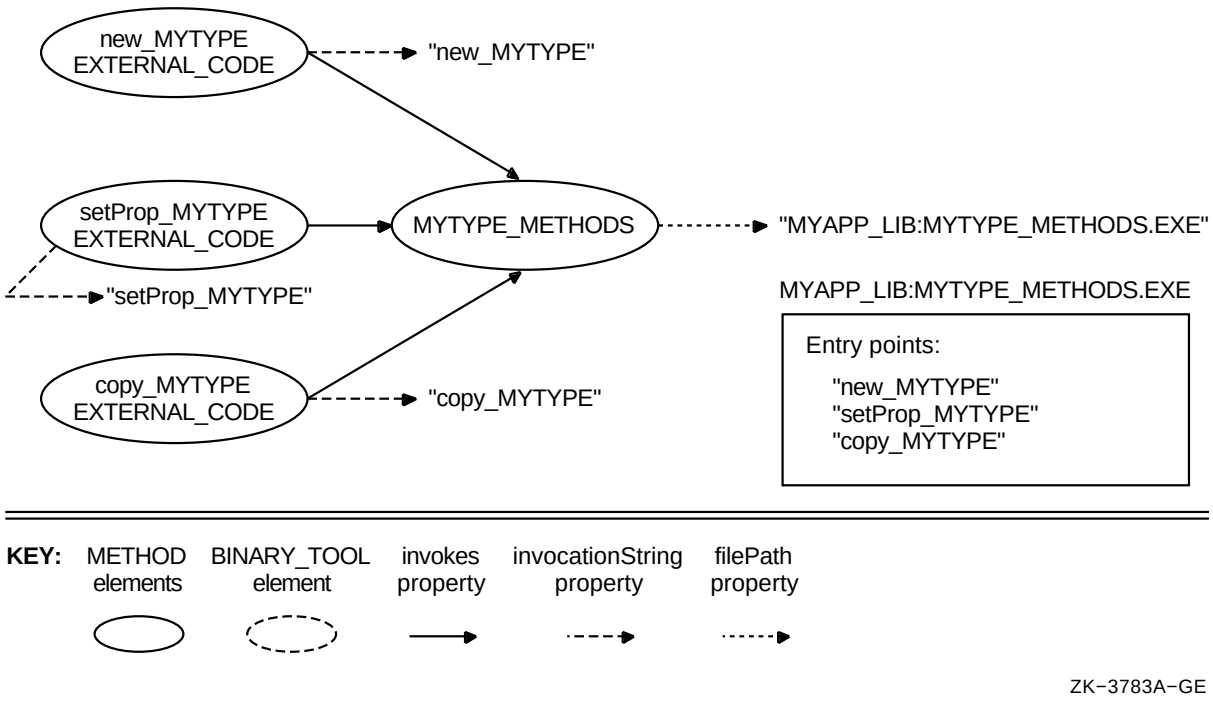
5.6.4 Associating the Element METHOD with External Code Methods

External code method functions are packaged in OpenVMS shareable images. To identify an entry point in a shareable image, you must supply the file specification of the shareable image and the name of the entry point. Oracle CDD/Repository provides for this information as follows:

- The **invocationString** property of the METHOD element contains the name of the entry point.
- The **invokes** property of the METHOD element contains the element ID of a BINARY_TOOL element. The **filePath** property of the BINARY_TOOL contains the file specification of the shareable image.

Figure 5–2 illustrates these connections for three external code methods.

Figure 5–2 External Code Methods and Entry Points



ZK–3783A–GE

Your application integration code must create the appropriate BINARY_TOOL and METHOD elements and set their property values appropriately to establish this association. If the value of the **filePath** property is an incomplete file specification, Oracle CDD/Repository supplies SYS$SHARE: for the device /directory and .EXE for the file type.

See the *OpenVMS* documentation set for information on creating shareable images.

5.7 Refining External Program Code Methods

An external program method is a program or command script. Unlike an external code method, an external program method executes as a separate process, not in the transaction or session in which it was invoked. If an external program method accesses the repository, it is responsible for all session and transaction operations.

When a message invokes a method whose function type is EXTERNAL_PROGRAM, the method dispatcher forms a command line to invoke the program or script. To form the command line, the method dispatcher uses the string contained in the **invocationString** property of the METHOD element (supplied by the application integrator) and substitutes values from the message. This process is detailed in Sections 5.7.1 through 5.7.6.

The program or script that implements an external program method is invoked by Oracle CDD/Repository calling LIB$SPAWN and passing a command line to run the program or execute a command file.

The process in which Oracle CDD/Repository runs continues while the external program method runs, but the transaction in which the method is called does not complete until the external program method completes. Because of this, if you use an external program method it may cause a transaction to be active for a long time. This in turn may cause locking problems in the repository because a transaction from the external program method and the transaction from the method that invoked it may try to access the same data and lock each other out.

To avoid this situation, you should not access the repository from external program methods and you should keep external program methods short. Use external program methods only to initiate other applications that can run concurrently with Oracle CDD/Repository. This technique allows your external program method to complete quickly and return control to the method that invoked it.

5.7.1 Invocation Strings

To create an external program method, you must write the invocation string for that method. This string, the value of the **invocationString** property on the METHOD element, can include placeholders for values to be substituted. When the method dispatcher has substituted the values in the invocation string, the resulting command line is used as the argument to LIB$SPAWN.

An invocation string can include values from a number of sources, including:

- arguments in the dispatch list sent with the message
- properties of the element to which the message was sent, or of any element that can be located starting with the target element
- properties of the current persistent process and context

Because different operating systems have different command-line formats, you must write an invocation string for each operating system on which your application operates.

5.7.2 Invocation String Syntax

An invocation string includes characters that are passed to the command line without alteration, and also may include zero or more token strings. Oracle CDD/Repository evaluates token strings and substitutes values in their place. The contents of the token string dictate what value is substituted.

The final value for any token string must be able to be represented as a string.

Table 5–1 summarizes invocation string syntax. The following sections provide more information and examples of various types of token strings.

Table 5–1 Invocation String Syntax Summary

Item	Meaning
%	Begins a token string.
%%	Specifies a literal percent sign in an unevaluated portion of the invocation string.
{ *space* | *tab* | *end-of-string* | % }	Terminates a token string. If %, begins the next token string.
->	Separates tokens in a token string.
%instance	Element ID of message target: represents the element ID of the element to which the message was sent. The *instance* keyword must follow the percent sign that starts the token string.
%instance->*property-name*	Property value: a token whose value is an element ID followed by a property name evaluates to the value of that property. If the property value is an element ID, another property name may be used to get the value of a property on the second element.
%argument->*argument-name*	Argument value from the dispatch list: the keyword *argument* followed by an argument name evaluates to the value of that argument. The *argument* keyword must follow the percent sign that starts the token string.
%argument->MCS_arglist-> argument->*argument-name*	Argument value from an argument list contained in the dispatch list.

(continued on next page)

Table 5–1 (Cont.) Invocation String Syntax Summary

Item	Meaning
%type-><*type-name*-> *element-name*	Element ID of an element: the keyword *type*, followed by a type name, followed by an element name evaluates to the element ID of the element of that type with that name. The *type* keyword must follow the percent sign that starts the token string.
list-valued-token-string->size	List size: when a token string evaluates to a property value that is a list, the *size* keyword returns the number of entries in the list.
list-valued-token-string->*n*	List entry: the value of the *n*th entry in a list.
list-valued-token-string->*	All list entries: a string containing the space-separated values of all the list entries.
memblock-valued-token-string-> length	Memblock length: when a token string evaluates to a property value that is a memblock, the *length* keyword returns the number of bytes in the memblock.
scan-valued-token-string->*-> *property-name*	Scan member property values: when a token string evaluates to a property value that is a scan, the * operator returns the element IDs of all the scan members. When a property name follows the asterisk, the token string evaluates to a string containing the space-separated values of that property for each member of the scan.
%currContext	Element ID of current CONTEXT element. The *currContext* keyword must follow the percent sign that starts the token string.
%currPersistentProcess	Element ID of current PERSISTENT_PROCESS element. The *currPersistentProcess* keyword must follow the percent sign that starts the token string.
%currDbElement	Element ID of DATABASE element currently associated with the element to which the message was sent. The *currDbElement* keyword must follow the percent sign that starts the token string.

Keywords are not case-sensitive.

Supply the name of a property, argument, element type, or element in an invocation string, not the symbol bound to that name. For example, to specify the **funcType** property, use *MCS_funcType* in the invocation string, not *MCS$r_prop_funcType* or *funcType*.

5.7.3 Substituting Dispatch List Arguments

When a token string begins with the *argument* keyword, followed by the name of an argument in the dispatch list, the value of that argument is substituted for the token string. For example, the **rename** message requires an argument, *MCS_new_name*, that specifies the new name of the element. The following token string evaluates to the new name value:

```
%argument->MCS_new_name
```

You also can use values from an argument list contained in the dispatch list. The argument list is identified by the argument name *MCS_arglist*. This means the following token string evaluates to the embedded argument list:

```
%argument->MCS_arglist
```

To extract a value from this argument list, use the *argument* keyword again. For example, assume the argument list contains the argument *MCS_funcType*. The following token string evaluates to the value of the *MCS_funcType* argument:

```
%argument->MCS_arglist->argument->MCS_funcType
```

If there is no argument with the name you specify after the *argument* keyword, the method dispatch fails.

You can use dispatch list arguments to pass arbitrary values to an external program method with the message that invokes the method. For example, you can include the *MCS_arglist* argument in the dispatch list of a message that does not usually take an argument list. The Oracle CDD/Repository method dispatcher ignores the argument list (because it is not listed as a required or optional argument) but the invocation string can still extract values from it.

5.7.4 Substituting Property Values

Whenever a token string evaluates to an element ID, you can append a property name to the token string to return the value of the property. Token strings that evaluate to element IDs include the following:

- The *instance* keyword, which evaluates to the element ID of the target element of the message. The following example shows how *instance* evaluates to the name of the element to which the message was sent:

  ```
  %instance->MCS_name
  ```

- An argument whose value is an element ID, taken from the dispatch list. The following example shows how an argument evaluates to the version number of the *merge* version in a **merge** message:

  ```
  %argument->MCS_merge_elmID->MCS_versionNum
  ```

- Arbitrary repository elements, identified by type and name. The following example shows how an element evaluates to the reservation status of the MESSAGE element named **open**:

  ```
  %type->MCS_MESSAGE->MCS_open->MCS_status
  ```

- A property whose value is an element ID. The following example shows how this string evaluates to the replacement time of the first version on the line of descent containing the message target:

  ```
  %instance->MCS_firstVersion->MCS_freezeTime
  ```

- The *currContext* and *currPersistentProcess* keywords, which evaluate to the element IDs of the current context and current persistent process, respectively. The following example shows how *currContext* evaluates to the name of the current context directory:

  ```
  %currContext->MCS_contextDir
  ```

  The following example shows how *currPersistentProcess* evaluates to the name of the current default directory.

  ```
  %currPersistentProcess->MCS_defaultDirectory
  ```

5.7.5 Substituting Values from Structured Data Types

When a token string evaluates to a list, a scan, or a memblock, you can substitute information from (or about) the value in the resulting command line.

5.7.5.1 Substituting Values from Lists

The *size* keyword following a token string that evaluates to a list returns the number of entries in the list. For example, the following token string evaluates to the number of symbols listed for the current persistent process:

```
%currPersistentProcess->MCS_symbols->size
```

An integer following a list-valued token string returns the value of the indicated list entry. For example, the following token string evaluates to the third symbol listed for the current persistent process:

```
%currPersistentProcess->MCS_symbols->2
```

An asterisk (*) returns a string containing the space-separated values of all list entries. The following token string evaluates to a string with all the symbols listed for the current persistent process:

```
%currPersistentProcess->MCS_symbols->*
```

5.7.5.2 Substituting Values from Scans

A token string that evaluates to a scan value, followed by an asterisk (*) and a property name, evaluates to a string containing the space-separated values of the specified property for all the elements in the scan. For example:

```
%type->MCS_ELEMENT_TYPE->MCS_VERSION->MCS_methods->*->MCS_name
```

This token string evaluates to a string with the names of all methods that operate on VERSION elements. (The scan in this token string is the value of the **methods** property on the ELEMENT_TYPE element named **MCS_VERSION**.)

5.7.5.3 Substituting the Length of a Memblock

A token string that evaluates to a memblock, followed by the *length* keyword, returns the length of the memblock.

5.7.6 Representation of Substituted Values

Oracle CDD/Repository uses the *MCS_datatype_read* routine to convert the value of a token string to a character string for use in the command line. See the *Oracle CDD/Repository Information Model Volume I* and *Oracle CDD/Repository Information Model Volume II* manuals for a list of the data types that can be converted to a string representation.

5.7.7 Associating External Program Methods with Files

The mechanism for associating an external program method with the file that implements the method is similar to the mechanism for external code methods (described in Section 5.6.4). That is:

- The **invokes** property of the METHOD element contains the element ID of a BINARY_TOOL or TEXT_TOOL element.
- The BINARY_TOOL or the **invokes** property of the TEXT_TOOL element property contains the file specification for the file that implements the method.

However, this information is used only for purposes of configuration management. The command line derived from the invocation string must contain all the information needed to execute the program.

6
Defining Properties

An element type automatically inherits properties possessed by its supertype, which include properties the supertype inherits. To these inherited properties, you can add properties that the new type defines.

The value of the **propDef** property of an ELEMENT_TYPE element specifies the properties that the type defines and inherits. Specifically, the value of **propDef** is a scan of **PROPERTY_TYPE** elements, each of which identifies a property.

To establish the set of properties defined by a type, insert the element IDs of PROPERTY_TYPE elements into the **propDef** scan. For normal properties, this procedure is relatively straightforward. For relation or computed properties, you must create additional elements and relationships.

You must create a PROPERTY_TYPE element for each property you want to define. Properties on the PROPERTY_TYPE element define specific characteristics, as follows:

- **name**—Defines the name of the new property. The **name** property is required if you create the element.
- **dataType**—Defines the data type of the property.
- **length**—Specifies the length of normal properties.
- **scale**—Specifies the scale factor for numeric property values.
- **accessType**—Specifies how the property may be accessed with **setProp**. Properties can be read only, read write, write once, or write once at creation.

Note

For better performance, always use MCS properties. Do not use CDD$ properties.

See the *Oracle CDD/Repository Information Model Volume I* and *Oracle CDD/Repository Information Model Volume II* manuals for information on the values each property can have.

6.1 Normal Properties

For a normal property (data is stored with the property), create the PROPERTY_TYPE element with appropriate values and insert its element ID into the **propDef** scan. Inserting the element ID in the scan creates a HAS_PROPERTY relationship between the ELEMENT_TYPE element and the PROPERTY_TYPE element. Make sure you define the **required** property for HAS_PROPERTY by using *MCS_scan_insert_with_args*.

6.2 Relation and Closure Properties

Relation and closure properties are implemented in a more complex fashion than normal properties.

- A HAS_RELATION_PROPERTY relationship connects the ELEMENT_TYPE element to the PROPERTY_TYPE element. The **relPropDef** property on an ELEMENT_TYPE element is a scan of properties connected via HAS_RELATION_PROPERTY. The value of **relPropDef** is a subset of the value of **propDef**, because HAS_RELATION_PROPERTY is a subtype of HAS_PROPERTY.
- The HAS_RELATION_PROPERTY instance has two properties that contain information about the relationships traversed by the relation property. One, **implementsRelation**, contains the element ID of the RELATION_TYPE element that defines the relationships. The other, **direction**, specifies if the relationships are traversed from owner to member or from member to owner. For closure properties, the direction is to all owners or to all members.

  Both these properties are required if you create new instances of HAS_RELATION_PROPERTY.
- A RELATION_TYPE element specifies the characteristics of the relationships traversed by the relation property. If you need relationships with characteristics not present in any types defined by Oracle CDD/Repository, you must create your own RELATION_TYPE instance.

To define a relation or closure property, you must construct the implementation for the property. Use the following steps:

1. Create (or reserve) the ELEMENT_TYPE element that represents the type you are defining.

2. Create a PROPERTY_TYPE element, just as you would for a normal property. The value of the **dataType** property for this element is MCS_ELEMENTID if the relation property being implemented always has a single element ID as its value, or MCS_SCAN if the property value can consist of more than one element ID.

3. Define a new RELATION_TYPE element (only if you need to set the owner and member properties). **MSC_legalOwner** and **MSC_legalMember** properties are not inherited.

4. Use the *MCS_scan_insert_with_args* routine to insert the element ID of the PROPERTY_TYPE element you created into the **relPropDef** property of the ELEMENT_TYPE element you created. Do this by sending **getProp** to get the value of **relPropDef**, calling *MCS_scan_insert_with_args*, then using **setProp** to set **relPropDef**.

   Inserting into **relPropDef** creates a HAS_RELATION_PROPERTY relationship. (If you insert into **propDef**, a HAS_PROPERTY relationship is created instead, which causes problems if you try to use it.)

   The arguments you supply with *MCS_scan_insert_with_args* initialize the **MSC_direction** and **MSC_implementsRelation** properties on the HAS_RELATION_PROPERTY relationship. Set **direction** to the appropriate value. (See the description of **direction** in the *Oracle CDD/Repository Information Model Volume I* manual for the proper constant values.) Set **implementsRelation** to the element ID of the RELATION_TYPE element that you identified or created earlier.

6.3 Defining Computed Properties

The property value must be computed by a method in response to a GETPROP message if a property value cannot be the following:

- stored with the element (a *normal* property)
- determined by traversing all relationships of a specified type (*relation* and *closure* properties)

Oracle CDD/Repository contains many examples of *computed* properties, including:

- The **numChildren** property on COMPOSITE elements, whose value is the number of children attached to the composite.
- The **lastVersion** property on VERSION elements, whose value is the element ID of the most recent version on a line of descent. This value must be computed by following the line of descent to the end.

- The **definedPropDefs** property on ELEMENT_TYPE, whose value is the subset of the properties contained in the **propDef** scan that are defined by the element type possessing the property. **definedPropDefs** is an example of a computed scan property (refer to Section 6.3.1).

These properties, along with all other computed properties, have the following characteristics:

- Their values can be derived algorithmically, but cannot be stored (because they capture a time-varying system state) or found by simple relationship traversal. If these values were stored, they would duplicate information that already exists in a different form; this is an inefficient practice and introduces the possibility of inconsistency.
- Their values cannot be set in a meaningful way. Instead, their values change as the result of other operations; for example, the value of **numChildren** changes as versions are attached to and detached from the composite.
- They are read only.

If Oracle CDD/Repository receives the GETPROP message for a computed property, it invokes the method associated with the property. This method receives the dispatch list and the element ID of the element that received the message. The method must perform the following functions:

1. Calculate the correct value for the property.
2. Insert the computed value in the appropriate argument specifier on the argument list.
3. Insert the MCS$_SUCCESS status code in the status field of the argument specifier.
4. Return a SUCCESS status.

The computation depends on the definition of the property, which depends on the needs of the application that defined the property.

6.3.1 Defining Computed Properties Whose Value Is a Scan

A computed scan property has a value that cannot be derived simply by traversing all relationships of a certain type. A computed scan may be a subset of a conventional scan, or it may consist of parts from several scans. A scan that is the value of a computed scan property must behave as if it were a conventional scan, including responding properly to *MCS_scan_getNext* calls and maintaining a current position.

A scan starts when it is returned by a GETPROP message and ends when *MCS_datatype_free* or *MCS_scan_free* is called on it. During this time, the scan may be the argument for any of the scan routines that read, remove, or add element IDs to it, and the scan also may become the new value of a property through SETPROP.

Because computed scan properties are read only, the operations that affect these scans are more restricted, as follows:

- They can be returned as the result of a GETPROP message.
- They can have element IDs returned from them by *MCS_scan_getFirst*, *MCS_scan_getNext*, and *MCS_scan_getCurrent* calls.
- They can be freed by a call to *MCS_datatype_free* or *MCS_scan_free*.

Only the first of these operations (return with GETPROP) is performed directly by the method that responds to the GETPROP message. Therefore, this method must establish the mechanism that allows the scan to handle the remaining operations. To set up this mechanism, the method calls the *MCS_scan_new* routine, passing the following two arguments, among others:

- The *scan_comp_rtn* argument is the address of a scan computation routine that Oracle CDD/Repository calls if the application calls *MCS_scan_getFirst*, *MCS_scan_getNext*, *MCS_scan_free*, or *MCS_datatype_free* on the computed scan.
- The *method_ctx* argument is the address of a data structure allocated and initialized by the method. This address is passed in turn by Oracle CDD/Repository to the scan computation routine, which can examine and store data needed to return the requested information and to maintain the scan context.

It is the job of the scan computation routine to compute and return the correct element ID for calls to *MCS_scan_getFirst* and *MCS_scan_getNext*, and to free allocated memory for a call to *MSC_scan_free* or *MCS_datatype_free*. If Oracle CDD/Repository calls the scan computation routine, it must pass a *scan_action* argument. The *scan_action* argument is a constant that indicates the required action. (See Section 6.3.2 for more information on the scan computation routine.)

To create a scan for a computed property, use the following steps:

1. Locally declare a value structure to contain the new scan.
2. Call *MCS_scan_new* to initialize the value structure to contain an empty scan. Your program specifies the following:
   - The value structure to contain the new scan.

- The element ID of the element that owns this property.
- The entry point to the routine that computes the value of the scan.
- The name of the property for which this scan is the value. You cannot create a scan that is not meant to be the value of a specific property.
- The address of a data structure containing context information the scan needs.

Do not specify an element ID for the type that owns the property.

Example 6–1 illustrates the implementation of a computed scan property. The property, **childrenWithA**, is defined by MY_COLL, a subtype of COLLECTION. (This example is for demonstration purposes only. Both the type and the property are imaginary. Also, this function is best implemented by calling *MCS_scan_getByName*.) The value of **childrenWithA** consists of those children of a MY_COLL instance whose name contains the letter *A*.

Example 6–1 Implementing a Computed Scan Property

```
typedef struct CHILDREN_WITH_A_STATE_INFO { ❶
    MCS_VALUE       childrenScan;
    MCS_BOOLEAN     firstTime;
    MCS_BOOLEAN     childScanExists;
    } ;

MCS_STATUS GET_NEXT_childrenWithA (); ❷

MCS_STATUS GETPROP_MY_COLL_childrenWithA ( ❸
    MCS_ELEMENTID   *instance,
    MCS_VALUE       *dispatchList
    )
{
MCS_STATUS          status, argStatus;
MCS_VALUE           propertyList, newScanValue;
MCS_LONGINT         proplistIndex;
CHILDREN_WITH_A_STATE_INFO *stateInfo;

    if ( ( status = MCS_arglist_findArg ( ❹
        dispatchList,
        MCS_arg_arglist,
        &propertyList,
        &argStatus,
        &proplistIndex ) ) != MCS_SUCCESS ) {
            return ( status );
        }
```

(continued on next page)

Example 6–1 (Cont.) Implementing a Computed Scan Property

```
stateInfo = (CHILDREN_WITH_A_STATE_INFO *)malloc( ❺
    sizeof(CHILDREN_WITH_A_STATE_INFO) );

if ( !stateInfo ) { ❻
    status = MCS_BADMALLOC;
    MCS_errorstack_set(status,0,0);
    MCS_arglist_setNameValue(
        &propertyList,
        "childrenWithA",
        0,
        status);
    return ( status );
    }

stateInfo->firstTime = TRUE; ❼
stateInfo->childScanExists = FALSE;

if ( ( status = MCS_scan_new ( ❽
    &newScanValue,
    instance,
    GET_NEXT_childrenWithA, ❾
    "childrenWithA",
    0,
    (void *)stateInfo ❿) ) != MCS_SUCCESS ) { ⓫
        MCS_arglist_setNameValue (
            &propertyList,
            "childrenWithA",
            0,
            status);
        free ( stateInfo );
        return ( status );
        }

if ( ( status = MCS_arglist_setNameValue ( ⓬
    &propertyList,
    "childrenWithA",
    &newScanValue,
    MCS_SUCCESS ) ) != MCS_SUCCESS ) {
        MCS_arglist_setNameValue (
            &propertyList,
            "childrenWithA",
            0,
            status);
        free ( stateInfo );
        return ( status );
        }
```

(continued on next page)

Example 6–1 (Cont.) Implementing a Computed Scan Property

```
    return ( MCS_SUCCESS );
}

MCS_STATUS GET_NEXT_childrenWithA ( ⓭
    MCS_ELEMENTID   *instance,
    void            *stateInfo, ⓮
    MCS_ELEMENTID   *result_elmID,
    MCS_ELEMENTID   *result_rID,
    MCS_SMALLINT    *scanAction )

{
    MCS_STATUS       status, getNameStatus;
    MCS_STRING       memberName;
    char             *lettersA = "Aa";

    switch ( *scanAction ) { ⓯
        case MCS_COMPUTED_SCAN_NEXT:
            if ( stateInfo->firstTime ) { ⓰
                /* Code to get the value of instance's
                   hasChildren scan into
                   stateInfo->childrenScan not shown.
                */
                stateInfo->childScanExists = TRUE;
                stateInfo->firstTime = FALSE;
                }
            status = MCS_scan_getNext ( ⓱
                &stateInfo->childrenScan,
                result_elmID,
                result_rID );
            while ( status == MCS_SUCCESS ) {
                if ( ( getNameStatus = MCS_element_getName (
                    result_elmID,
                    memberName ) ) != MCS_SUCCESS ) {
                        return ( getNameStatus );
                    }
                if ( strpbrk (
                    memberName,
                    lettersA ) ) {
                        free ( memberName ); ⓲
                        return ( MCS_SUCCESS );
                    }
                free ( memberName );
                status = MCS_scan_getNext (
                    &stateInfo->childrenScan,
                    result_elmID,
                    result_rID );
                }
```

(continued on next page)

Example 6–1 (Cont.) Implementing a Computed Scan Property

```
            return ( status );

        case MCS_COMPUTED_SCAN_RESET: ⓳
            if ( stateInfo->childScanExists ) {
                if ( ( status = MCS_scan_free (
                    &stateInfo->childrenScan ) )
                      != MCS_SUCCESS ) {
                    return ( status );
                    }
                stateInfo->childScanExists = FALSE;
                stateInfo->firstTime = TRUE;
                }
            return ( MCS_SUCCESS );

        case MCS_COMPUTED_SCAN_FREE: ⓴
            if ( stateInfo->childScanExists ) {
                if ( ( status = MCS_scan_free (
                    &stateInfo->childrenScan ) )
                      != MCS_SUCCESS ) {
                    free ( stateInfo );
                    return ( status );
                    }
                }
            free ( stateInfo );
            return ( MCS_SUCCESS );

        default: ㉑
            return ( MCS_BADPARAM );

        }
}
```

Key to Example 6–1:

❶ Each **childrenWithA** scan has an associated data structure that keeps track of context information for the scan. This is the declaration of that structure. The childrenScan field stores the value of **hasChildren** for the MY_COLL element; the firstTime and childScanExists fields record information about prior usage of the structure.

❷ Forward declaration of the scan computation routine.

❸ Beginning of the method function. This is called once by Oracle CDD/Repository when an application sends GETPROP to get the value of **childrenWithA**.

❹ Obtains the property list (containing the list of properties to be gotten) from the message dispatch list.

❺ Allocates a CHILDREN_WITH_A_STATE_INFO structure and assigns its address to stateInfo.

❻ If the structure allocation failed, inserts the failure status in the argument spec and returns the same status.

❼ Initializes the structure to indicate that the scan has not been used yet (that is, has not had an *MCS_scan_get** operation called on it) and that the scan of collection children does not yet exist.

❽ The call to *MCS_scan_new* creates the scan value for **childrenWithA**.

❾ The address of the scan computation routine.

❿ The address of the CHILDREN_WITH_A_STATE_INFO structure.

⓫ If the call to *MCS_scan_new* fails, put the failure code in the argument spec, free the data structure, and return the failure code.

⓬ Place the newly created scan in the property list, set the status field to SUCCESS, and return.

⓭ Beginning of the scan computation routine. This routine is called by Oracle CDD/Repository when the application calls *MCS_scan_getNext*, *MCS_scan_getFirst*, *MCS_scan_free*, or *MCS_datatype_free* on a **childrenWithA** value. (See Chapter 8 for more information.)

⓮ The address of the data structure that contains context information for this scan value.

⓯ The scanAction argument specifies the action to be performed by the scan computation routine. The actions result from scan calls, as follows:

MCS_scan_getNext: NEXT
MCS_scan_getFirst: RESET, then NEXT
scan_free or datatype_free: FREE

⓰ If the scan has not been accessed before, this code (some not shown) gets the value of the **hasChildren** property of the MY_COLL element. The value of **childrenWithA** is a subset of this scan. Fields in the data structure are set to indicate that the **hasChildren** scan is now present and that the **childrenWithA** value has been accessed.

⓱ Start from the current position of the **hasChildren** scan and get the name of successive elements until one containing an *A* is found or until the end of the scan is reached. This code returns MCS_SUCCESS if an element name containing *A* is found, MCS_ENDFIND if the name is not found, or some other status code reflecting a problem with the *MCS_scan_getNext* call. The calls to *MCS_scan_getNext* use the *result_elmID* and *result_rID*

arguments passed to the scan computation routine, which makes sure they are set correctly if the routine returns SUCCESS.

⓲ Because *MCS_element_getName* allocates space for the name string, you must be sure to free the space before returning.

⓳ The response to the RESET action is to restore initial conditions. This requires freeing the **hasChildren** scan if it exists and resetting flags in the data structure to their initial conditions.

⓴ The FREE action requires that all memory be freed. This code frees the **hasChildren** scan and the context data structure, and returns.

㉑ If the value of the scanAction is other than one of the three values in the case clauses, there is some type of internal problem.

You do not need to write a GETPROP method for properties whose implementation type is *normal* or *relation*. The definition of the property supplies enough information to allow Oracle CDD/Repository to determine these values without additional assistance.

However, computed properties are implemented by defining a method that computes the value of the property in response to the **getProp** message, as follows:

- Use a HAS_COMPUTED_PROPERTY relationship to connect the ELEMENT_TYPE element to the PROPERTY_TYPE element; do not use HAS_PROPERTY. The **compPropDef** property on an ELEMENT_TYPE element is a scan of properties connected via HAS_COMPUTED_PROPERTY. The value of **compPropDef** is a subset of the value of **propDef**, because HAS_COMPUTED_PROPERTY is a subtype of HAS_PROPERTY.
- Each HAS_COMPUTED_PROPERTY instance has the **implementsMethod** property. The value of this property is the element ID of the METHOD element that computes the property value. This property is required if you create a new instance of HAS_COMPUTED_PROPERTY.
- The method that computes the property receives the argument list that was provided by the sender of the **getProp** message. The method computes the property value and fills it in as the value of the property in the argument list. It only responds to **getProp**, not to **setProp**.

The steps required to implement a computed property are similar to those required to implement a relation or closure property.

1. Create the ELEMENT_TYPE element.

2. Create the PROPERTY_TYPE element. Initialize its **dataType** property to the appropriate value, depending on the value that the method computes. Initialize **accessType** to read only.

3. Create the METHOD element that represents the computation of the value of the property. The method is responsible for filling in the appropriate argument specifier in the argument list that it receives with the computed value. The method does not need to be attached to any message; it implicitly responds to **getProp**.

4. Use the *MCS_scan_insert_with_args* routine to insert the element ID of the PROPERTY_TYPE element you created into the **compPropDef** property of the ELEMENT_TYPE element you created. Do this by sending **getProp** to get the value of **compPropDef**, calling *MCS_scan_insert_with_args*, then using **setProp** to set **compPropDef**. This procedure creates a HAS_COMPUTED_PROPERTY relationship that connects the ELEMENT_TYPE and PROPERTY_TYPE elements.

   The argument you supply with *MCS_scan_insert_with_args* initializes the **implementsMethod** property on the HAS_COMPUTED_PROPERTY relationship. Supply the element ID of the METHOD element you created as the value for **implementsMethod**.

(Chapter 5 explains method writing and method refinement.)

6.3.2 Coding a Scan Computation Routine

If your application defines a computed property, the application must designate a method to compute the value of the property. The method invokes the *method computation routine*. For computed properties whose values are not scans, the method computation routine computes the value, places it in the argument list that was passed to it, and exits.

Scans are different. The application can traverse the scan with *MCS_scan_getNext*, reset it with *MCS_scan_reset*, and free it with *MCS_scan_free*. A computed scan property must respond appropriately to each of these calls.

The method computation routine is called only once, in response to the GETPROP message that gets the value of the computed scan property. It calls *MCS_scan_new* to establish the computed scan for future access by *MCS_scan_getNext* and other scan calls. The call to *MCS_scan_new* must therefore define how to respond to those calls. It does this by passing the address of a *scan computation routine*. When scan calls are subsequently made for that scan, Oracle CDD/Repository calls the scan computation routine, which is responsible for determining and returning the appropriate element ID to return for the specified operation. The scan computation routine must also return an appropriate status to the caller.

The scan computation routine is called with the following arguments:

element_elementid
type: MCS$R_ELEMENTID
access: read only
mechanism: by reference

Element ID of the element that owns the computed scan property.

method_ctx
type: unspecified
access: read/write
mechanism: by reference

Pointer to an area of context information maintained for the use of the scan computation routine.

result_elementid
type: MCS$R_ELEMENTID
access: write only
mechanism: by reference

Element ID of the appropriate element for the specified operation. This becomes the value of the *element_elementid* argument for the scan call.

relation_elementid
type: MCS$R_ELEMENTID
access: write only
mechanism: by reference

Element ID of the relationship that connects *element_elementid* to *result_elementid*. This becomes the value of the *relation_elementid* argument for the scan call.

scan_action
type: MCS$W_SMALLINT
access: read only
mechanism: by reference

Action to be performed by the scan computation routine.

Possible values are listed in the following table:

C Binding	OpenVMS Binding	Action
MCS_COMPUTED_ SCAN_NEXT	MCS$K_COMPUTED_ SCAN_NEXT	Returns the element following the current position. If the scan has never been accessed or has just been reset, returns the first element in the scan. If the current position is the last item in the scan, returns ENDFIND as the status of the scan computation routine.
MCS_COMPUTED_ SCAN_RESET	MCS$K_COMPUTED_ SCAN_RESET	Resets the scan to its initial state; do not fill in the *element_elementid* or *relation_elementid* arguments.
MCS_COMPUTED_ SCAN_FREE	MCS$K_COMPUTED_ SCAN_FREE	Deallocates any data structures (such as the method context) that supported computation of the scan's value.

A call to *MCS_scan_getFirst* results in two calls to the scan computation routine: the first with an action of RESET, and the second with an action of NEXT.

The method computation routine must also allocate and initialize a data structure to hold context information for the computed scan, and pass its address as the *method_ctx* argument to *MCS_scan_new*. The scan computation routine receives the address of this data structure. It can store control information in the data structure to maintain the scan context from one scan access call to the next.

The scan computation routine must return an MCS status code, as shown in the following table:

Status	Meaning
SUCCESS	The scan computation routine completed successfully. For a SCAN_NEXT request, an element was identified and the *result_elementid* and *relation_elementid* arguments were filled in.
ENDFIND	The scan computation routine completed a SCAN_NEXT request without error, but no next element was identified.
Other MCS status code	The scan computation routine encountered an error.

6.4 Refining the setProp and new Methods

Two messages explicitly set property values: **setProp**, and **new**. These messages supply initial values for properties on the new element. You do not have to set the property value. Oracle CDD/Repository sets the value without assistance. However, you may want to write refinements for **setProp** or **new** to check that a property value is in a specified range or to perform side effects associated with setting the value of the property.

For example, you define a new element type, X_TYPE, and a property for it, **yProp**. In the definition of the property, you specify the following information:

- The data type of the property. Use MCS_SMALLINT.
- The implementation type of the property. Implement **yProp** as a *normal* property (data is stored with the element).
- The access type of the property. Make **yProp** a *read-write* property, which means that its value can be changed with **setProp**.
- If the property must be set when creating new instances of X_TYPE. Specify that it does not have to be set.

However, if your application requires that the value of **yProp** be one of a set of predefined constant values, you cannot specify this requirement in the repository definition of **yProp**. To enforce the requirement, you must refine **setProp** and **new** for X_TYPE elements. The refinement checks that the value to be set is one of the predefined values. The refinement operates as follows:

- If the **yProp** property is not specified, there is nothing to check. The refinement calls *MCS_dispatch_superOp* to set the property values on the list.

  You can set more than one property with a call to **setProp** or **new**.
- If the property is specified and the supplied value is acceptable, the refinement calls *MCS_dispatch_superOp*, then returns. The method invoked by *MCS_dispatch_superOp* sets the property value, along with any other values on the list.
- If the value is not acceptable, the refinement does not call *MCS_dispatch_superOp*. Instead, it places an application-specific status code indicating that the value was out of range in the status field of the argument spec containing the property value, and pushes the same status code onto the error stack. It next pushes the status code MCS_SETPROPFAILED onto the error stack. Finally, the method refinement returns MCS_SETPROPFAILED.

If the refinement does not call *MCS_dispatch_superOp*, none of the properties specified in the **setProp** message are set. After receiving MCS_SETPROPFAILED, you can examine the status field of the argument specs in the property list to determine which property caused the problem. You also should abort the transaction in which **setProp** was sent, to be sure that any side effects of setting properties are undone.

Example 6–2 shows the refinement of **setProp** of the X_TYPE to enforce the value requirement. The refinement for **new** is similar.

Example 6–2 Refining setProp to Check a Value

```
/* Example of a refinement of setProp for the element type X_TYPE.
   Checks if the value of the yProp property is being set, and if
   so makes sure it is between YPROP_VALUE1 and YPROP_VALUE_LAST.
*/

MCS_STATUS SETPROP_X_TYPE (
    MCS_ELEMENTID   *instance,
    MCS_VALUE       *dispatchList
    )
{
MCS_STATUS          status, argStatus, dataStatus;
MCS_VALUE           propertyList,   /* List of props on dispatch list */
                    propertyValue;  /* Value struct of yProp property */
MCS_LONGINT         proplistIndex,  /* Index of property list argspec */
                    yPropIndex;     /* Index of yProp argspec */
MCS_SMALLINT        yPropValue;     /* Numeric value of yProp */
MCS_ELEMENTID       type_elmID, elementtype_elmID, XType_elmID;

    /* Get the property list from the dispatch list */
    if ( ( status = MCS_arglist_findArg (
        dispatchList,
        MCS_arg_arglist,
        &propertyList,
        &argStatus,
        &proplistIndex ) ) != MCS_SUCCESS ) {
            return ( status );
        }

    /* See if the yProp property is on the property list */
    status = MCS_arglist_findArg (
        &propertyList,
        "yProp",
        &propertyValue,
        &argStatus,
        &yPropIndex );
```

(continued on next page)

Example 6–2 (Cont.) Refining setProp to Check a Value

```
/* Act on the result of findArg */
switch ( status ) {
    /* If ARGNOTFOUND, yProp was not specified.
       Do the superop.  */
    case MCS_ARGNOTFOUND:
        break;

    /* If SUCCESS, yProp was specified.  Check its value. */
    case MCS_SUCCESS:
        if ( ( dataStatus = MCS_datatype_read (
            &propertyValue,
            MCS_datatype_smallint,
            &yPropValue ) ) != MCS_SUCCESS {
                /* Set the status on the yProp argspec */
                MCS_arglist_setIndexValue (
                    &propertyList,
                    yPropIndex,
                    &propertyValue,
                    dataStatus );
                return ( dataStatus );
            }
        if ( ( yPropValue >= YPROP_VALUE1 ) &&
             ( yPropValue <= YPROP_VALUE_LAST ) ) {
            break;          /* Value OK, do superop */
            }
        else {              /* Value out-of-bounds */
            /* Set the status on the yProp argspec */
            MCS_arglist_setIndexValue (
                &propertyList,
                yPropIndex,
                0,          /* Do not change the property value */
                YPROP_BADVALUE);
            /* Push error codes on error stack */
            MCS_errorstack_set (
                YPROP_BADVALUE, 0, 0);
            MCS_errorstack_set (
                MCS_SETPROPFAILED, 0, 0);
            /* Finally, return MCS_SETPROPFAILED */
            return ( MCS_SETPROPFAILED );
            }

    /* Unexpected code from arglist_findArg - return it */
    default:
        return ( status ) ;

    }  /* End of switch (status)  */
```

(continued on next page)

Example 6–2 (Cont.) Refining setProp to Check a Value

```
    /* If yProp is present and in range, or if it is not present,
       call the superop.  (See Section 5.6.3 for
       details of this process.)  */

    if ( ( status = MCS_element_getType (
        instance,
        &type_elmID ) ) != MCS_SUCCESS ) {
            return ( status );
        }
    if ( ( status = MCS_element_getType (
        &type_elmID,
        &elementtype_elmID ) ) != MCS_SUCCESS ) {
            return ( status );
        }
    if ( ( status = MCS_element_getByName (
        &elementtype_elmID,
        "X_TYPE",
        &XType_elmID,
        0 ) ) != MCS_SUCCESS ) {
            return ( status );
        }

    return (MCS_dispatch_superOp (
        &XType_elmID,
        instance,
        MCS_message_setProp,
        dispatchList ) );
}
```

6.4.1 Side Effects of Setting Property Values

In addition to the value checks illustrated in Example 6–2, **setProp** and **new** refinements may make changes in the system state to reflect the new value of the property. An element represents some object such as a file or system configuration. Changing an element with **setProp** may require a change to the object it represents. If this is the situation, the **setProp** refinement must make the necessary changes in the repository or elsewhere in the environment.

These types of side effects depend on the needs of the application. For examples from the element types defined by Oracle CDD/Repository, see the following methods:

- The SETPROP refinement for BINARY handles changes to the **fileName** property by renaming the file and symbolic links associated with the element.

- The SETPROP refinement for CONTEXT handles changes to the **top** property by deleting the directory structure corresponding to the old top collection and creating a new directory structure for the new top collection.
- The NEW refinement for BINARY verifies that certain combinations of properties are present and creates a delta file for the new element under certain circumstances.

A SETPROP or NEW refinement that operates on files outside the file system controlled by Oracle CDD/Repository should use the Oracle CDD/Repository file operation routines rather than native file system facilities. Should the transaction that sent the SETPROP or NEW message abort, file manipulations performed with the file operation routines are undone automatically. This allows your application to keep its private files in a consistent state with the repository.

7
Using Notices

This chapter includes the following topics:

- notice services
- when notices are sent
- the MCS_noticeAction property
- notice actions
- notice-processing calls
- NOTICE data type

7.1 Notice Services

Oracle CDD/Repository uses relationships to pass notices among dictionary elements. For example, if you create a new version of a field definition, Oracle CDD/Repository automatically sends a notice to any related record definitions. This notice tells the applications using that definition that someone has created a new version of a field definition used by this record definition.

7.2 When Notices Are Sent

If a version of an element that is a member of a relationship of the type DEPENDS_ON (or its subtypes) is changed or created, Oracle CDD/Repository notifies the owner of the element. Oracle CDD/Repository sends one of four notices, based on the nature of the change:

- MCS_POSSIBLY_INVALID

  The element receiving the notice might be invalid, due to a recent change to a related element.

- MCS_INVALID

  The element receiving the notice is definitely invalid because a related element has been changed or deleted.

- MCS_CHILD_USAGE

  The element receiving the notice might be invalid because a relationship whose owner is related to the element has been changed.

- MCS_NEW_VERSION

  The element receiving the notice is related to an element for which a new version has been created. That is, the element is related to a recently superseded element.

7.3 The MCS_noticeAction Property

Every relation type in the dictionary owns the MCS_noticeAction property. The value of this property controls how Oracle CDD/Repository transmits notices through relation instances.

There are three legal values (from the CDDTAGS.* language support files) for the MCS_noticeAction property:

- MCS_SUCCESS = 1

  If a new version of the element is created, Oracle CDD/Repository interprets a relationship value of MCS_SUCCESS as though it were MCS_ SIGNAL and sends a notice to the owner of the relationship. If the element is changed without creating a new version, MCS_SUCCESS indicates that the owner of the relationship does not receive a notice. However, Oracle CDD/Repository tries to send the notice to other elements that own the owner.

- MCS_SIGNAL = 2

  Oracle CDD/Repository sends a notice to the owner of the relationship with the MCS_SIGNAL value. It then continues sending notices to owners of the owner until a relationship with a notice action attribute of MCS_ BLOCK is encountered or there are no more elements related to the owner.

- MCS_BLOCK = 3

  Oracle CDD/Repository interprets this value the same way whether an element has been changed in place or a new version of an element has been created.

  The owner of a relationship with this value does not receive a notice and Oracle CDD/Repository makes no attempt to send notices to other elements that own the owner.

Note

Using CDO to modify an element_type, relation_type, or property_type is not recommended; you should use the Oracle CDD/Repository callable interface Version 5.*n* or higher.

7.4 Notice Actions After a Change

When an element has been changed, Oracle CDD/Repository interprets the values of the MCS_noticeAction property as follows:

- MCS_SUCCESS = 1

  The owner of a relationship with this value does not receive a notice, but Oracle CDD/Repository tries to send a notice to other elements that own the owner.

- MCS_SIGNAL = 2

  The owner of a relationship with this value receives a notice. Oracle CDD/Repository continues sending notices to owners of the owner until a relationship with a value of MCS_BLOCK is encountered or the owner has no owners. The MCS_SIGNAL value overrides the MCS_SUCCESS value.

- MCS_BLOCK = 3

  The owner of a relationship with this value does not receive a notice and Oracle CDD/Repository makes no attempt to send notices to other elements that own the owner.

In Table 7–1, A, B, C, D, E, and F are elements that are members of relationships. The value of the MCS_noticeAction property for each relationship is shown.

If a change is made to F, no notice is received on E because the relationship between F and E has a value of MCS_SUCCESS. Oracle CDD/Repository continues trying to send notices upward.

No notice is received on D because the relationship between E and D has a value of MCS_SUCCESS. Oracle CDD/Repository continues trying to send notices upward.

C receives a notice because the relationship between D and C has a value of MCS_SIGNAL.

The relationship between C and B has a value of MCS_SUCCESS, which means that normally B would not receive a notice. However, in this case, the value of MCS_SIGNAL on the relationship between D and C remains in effect, overriding the value of MCS_SUCCESS on the relationship between C and B. Therefore, B receives a notice.

The notices stop at B because the relationship between B and A has a value of MCS_BLOCK, which does not allow any notices to pass.

Table 7–1 Values of the MCS_noticeAction Property

Elements	Receives Notice for Change	Receives Notice for Creation	MCS_noticeAction Properties
A	No	No	
			MCS_BLOCK
B	Yes	Yes	
			MCS_SUCCESS
C	Yes	Yes	
			MCS_SIGNAL
D	No	Yes	
			MCS_SUCCESS
E	No	Yes	
			MCS_SUCCESS
F	No	No	

The MCS_noticeAction property exists on relationship protocol definitions, not relationship instances. Therefore, all instances of a particular relationship type have the same notice action.

7.5 Notice Actions for New Elements

When a new element has been created, Oracle CDD/Repository interprets the values of the MCS_noticeAction property as follows:

- MCS_SUCCESS = 1

  Oracle CDD/Repository interprets a relationship value of MCS_SUCCESS as though it were MCS_SIGNAL.

- MCS_SIGNAL = 2

The owner of a relationship with this value receives a notice. CDD/Repository continues sending notices to owners of the owner until a relationship with a notice action property of MCS_BLOCK is encountered or there are no more elements related to the owner.

- MCS_BLOCK = 3

  The owner of a relationship with this value does not receive a notice and Oracle CDD/Repository makes no attempt to send notices to other elements that own the owner.

Consider Table 7–1 again. A, B, C, D, E, and F are elements that are members of relationships. The value of the MCS_noticeAction property for each relationship is shown.

If a new version of F is created, notices are received on E, D, C, and B. This is because the relationships between F and E, E and D, D and C, and C and B have values of MCS_SIGNAL or MCS_SUCCESS. MCS_SIGNAL values cause notices to be received on the owner of the relationship, and when new elements are created, Oracle CDD/Repository interprets MCS_SUCCESS values as though they were MCS_SIGNAL values. No notices are sent to element A because the relationship between B and A has a value of MCS_BLOCK.

The MCS_noticeAction property exists on relationship protocol definitions, not relationship instances. Therefore, all instances of a particular relationship type have the same notice action.

7.6 Notice-Processing Calls

Oracle CDD/Repository supplies the following calls that help your program keep track of changes in the repository:

- *MCS_check_notices*—Reads the notices an element has received. Section 7.7 explains the notice data type.
- *MCS_clear_notices*—Removes old notices after you have read them.
- *MCS_force_notices*—Causes notices to propagate even if you have not changed an element.
- *MCS_read_notices*—Returns the element ID of the sender of the notice and the cause.

7.7 NOTICE Data Type

Oracle CDD/Repository defines the MCS_NOTICE type as a structure that contains information generated when one element changes and notifies another about the change. Some elements are related to others by dependency relationships. One element owns the relationship and the other element is a member of the relationship. The relationship owner depends on the member. If a member element changes (for example, if a property value is changed), it notifies its owner that it has changed. An instance of MCS_NOTICE represents this notification. Each instance of MCS_NOTICE contains the following information:

- element that received the notice
- element that sent the notice
- reason the notice was sent

You can use **MCS_read_notices** to get these values.

8
Routine Descriptions

Each routine description in this chapter is divided into sections with the following headings:

Routine Title

Contains the name of the routine. It is followed by one or two sentences that further describe the routine's use. The name in the title is the name used in the routine's C binding; if you are using the OpenVMS binding, you write the name differently.

OpenVMS Format

Provides the OpenVMS binding name for the routine and lists its arguments in order.

Arguments

Gives a detailed description for each argument listed in the OpenVMS Format section. The following information is provided for each argument:

- Type—The data type of the argument. The data type name is given in its OpenVMS binding.
- Access—The routine's access to the argument. Access types include the following:
  - Read only—The routine reads the contents of the argument but does not change the value of the argument.
  - Write only—The routine ignores the contents of the argument and stores a value in the argument.
  - Read/write—The routine reads the contents of the argument and may store a new value in the argument.

In an access description, the term *argument* includes the argument and everything it points to, either directly or indirectly. For example, a value structure can contain a pointer to other data. If the access for a value structure is listed as read only, the value structure must be initialized and must point to valid data. If the access is read/write, the value structure and data must be valid as above, and both may be changed by the routine. If the access is write only, the initial contents of the value structure are ignored and may be overwritten, destroying any pointers to data.

- Mechanism—The passing mechanism for the argument *in the OpenVMS binding*. All arguments are passed by reference, with the exception of character strings, which are passed by descriptor.
- Description—A brief description of the argument and its use.

Optional Arguments

If an argument's description starts with the word *Optional*, that argument may be omitted or you may request the default; the description includes information on the default. Request the default by passing a zero *by immediate value* for that argument.

C Binding

Contains the C binding for the routine in the form of a C function prototype. Examine this prototype to determine the C binding name for the data type of each argument and the passing mechanism for each argument. Find the remaining argument information in the Arguments section.

Description

Provides a detailed description of the routine's operation and briefly explains its uses. The description is not intended to serve as a tutorial.

Condition Values

Lists the routine's possible status return values and provides a brief description of each one. The values are given by their generic names, but you must test for them by using their binding symbols. For example:

Generic Name	C Binding Symbol	OpenVMS Binding Symbol
SUCCESS	MCS_SUCCESS	MCS$_SUCCESS

See Also

Lists other routines that are frequently used in conjunction with this routine or whose descriptions may provide further insight. An asterisk (*) at the end of a routine name indicates all routines that start with that name; for example, *MCS_scan_get** can indicate *MCS_scan_getFirst*, *MCS_scan_getNext*, or *MCS_scan_getByName*.

MCS_arglist_addArg

Builds an argspec and puts it in an argument list in a single operation.

OpenVMS Format

```
status = MCS$arglist_addArg   argument_name, [argument_value,] [argument_status,]
                              argument_list_value
```

Arguments

argument_name
type: MCS$R_STRINGDSC
access: read only
mechanism: by descriptor

Name to insert into the new argspec.

argument_value
type: MCS$R_VALUE
access: read only
mechanism: by reference

Optional. Value structure containing the value to insert into the new argspec. Following the operation, the argspec and this value structure will refer to the same memory. If you omit this argument, an argspec with a data type of MCS_UNSPECIFIED is created.

argument_status
type: MCS$L_STATUS
access: read only
mechanism: by reference

Optional. Initial status to insert into the new argspec. If you omit this argument, the argspec's status is undefined.

argument_list_value
type: MCS$R_VALUE
access: read/write
mechanism: by reference

Value structure pointing to the argument list into which to insert the new argspec.

C Binding

```
MCS_STATUS MCS_arglist_addArg (
                MCS_STRING argument_name,
                MCS_VALUE *argument_value,
                MCS_STATUS argument_status,
                MCS_VALUE *argument_list_value)
```

Description

The *MCS_arglist_addArg* routine builds an argspec and puts it in an argument list in a single operation. The list must have been created previously with MCS_list_new. The new argspec is added to the end of the list.

Following this call, the argspec in the argument list points to the actual data pointed to by the *argument_value* argument; therefore, do not call *MCS_list_free* or *MCS_datatype_free* on the *argument_value* argument, since to do so would corrupt the argument list.

Condition Values

ERRARGLIST_ADD	Error attempting to add the argspec; examine the error stack for the cause.
SUCCESS	Normal successful completion.

See Also

MCS_list_new

MCS_arglist_findArg

Locates an argspec in an argument list by the name of the argument it contains, and returns the value and status of the argspec.

OpenVMS Format

status = MCS$arglist_findArg argument_list_value, argument_name, [argument_value,] [argument_status,] index_number

Arguments

argument_list_value
type: MCS$R_VALUE
access: read only
mechanism: by reference

Value structure pointing to the list of argspecs.

argument_name
type: MCS$R_STRINGDSC
access: read only
mechanism: by descriptor

Name of argument.

argument_value
type: MCS$R_VALUE
access: write only
mechanism: by reference

Optional. Value structure; filled in with the value of the argument. The application is responsible for allocating the value structure.

argument_status
type: MCS$L_STATUS
access: write only
mechanism: by reference

Optional. Filled in with the status of the argument.

index_number
type: MCS$L_LONGINT
access: write only
mechanism: by reference

Optional. Filled in with the index of the argument in the argument list.

C Binding

```
MCS_STATUS MCS_arglist_findArg (
               MCS_VALUE *argument_list_value,
               MCS_STRING argument_name,
               MCS_VALUE *argument_value,
               MCS_STATUS *argument_status,
               MCS_LONGINT *index_number)
```

Description

The *MCS_arglist_findArg* routine returns the value, status, and index of an argument from an argument list, given its name.

The value structure returned in the *argument_value* argument contains a pointer to the actual data in the argument list. Therefore, do not call *MCS_list_free* or *MCS_datatype_free* on this value structure, since this will corrupt the argument list by destroying the data in that argument.

You can omit the last three arguments to this routine in order to determine whether or not an argument list contains a specified argument.

Condition Values

ARGNOTFOUND	The argument list did not contain the specified argument.
ERRARGLIST_FIND	Error attempting to find the argspec; examine the error stack for the cause.
SUCCESS	Normal successful completion.

See Also

MCS_arglist_getArg

MCS_arglist_getArg

Locates an argspec by its index into an argument list, and returns the argument value, status, and name.

OpenVMS Format

```
status = MCS$arglist_getArg  argument_list_value, index_number, [argument_value,]
                             [argument_status,] argument_name
```

Arguments

argument_list_value
type: MCS$R_VALUE
access: read only
mechanism: by reference

Value structure pointing to the list of argspecs.

index_number
type: MCS$L_LONGINT
access: read only
mechanism: by reference

Index of argument in list.

argument_value
type: MCS$R_VALUE
access: write only
mechanism: by reference

Optional. Value structure; filled in with the value of the argument. The application is responsible for allocating the value structure.

argument_status
type: MCS$L_LONGINT
access: write only
mechanism: by reference

Optional. Filled in with the status of the argument.

argument_name
type: MCS$R_STRINGDSC
access: write only
mechanism: by descriptor

Optional. Filled in with the name of the argument stored at this offset in the argument list. The descriptor supplied by the application must be uninitialized. The application is responsible for deallocating this memory after use.

C Binding

```
MCS_STATUS MCS_arglist_getArg (
              MCS_VALUE *argument_list_value,
              MCS_LONGINT index_number,
              MCS_VALUE *argument_value,
              MCS_STATUS *argument_status,
              MCS_STRING *argument_name)
```

Description

The *MCS_arglist_getArg* routine returns the value, status, and name of an argument on an argument list, given its index.

The value structure returned in the *argument_value* argument contains a pointer to the actual data in the argument list. Therefore, do not call *MCS_list_free* or *MCS_datatype_free* on this value structure, since this will corrupt the argument list by destroying the data in that argument.

Condition Values

ERRARGLIST_GET	Error attempting to get the argspec; examine the error stack for the cause.
SUCCESS	Normal successful completion.

See Also

MCS_arglist_findArg

MCS_arglist_setIndexValue

Modifies the value and status contained in the argspec at a specified index in an argument list.

OpenVMS Format

status = MCS$arglist_setIndexValue argument_list_value, index_number, [argument_value,] [argument_status]

Arguments

argument_list_value
type: MCS$R_VALUE
access: read/write
mechanism: by reference

Value structure pointing to the argument list.

index_number
type: MCS$L_LONGINT
access: read only
mechanism: by reference

Index of argspec to modify.

argument_value
type: MCS$R_VALUE
access: read only
mechanism: by reference

Optional. Value structure containing new value to set. Following the operation, the argspec and this value structure will refer to the same memory. If you omit this argument, the current value structure remains intact. This is useful if you want to set only the status.

argument_status
type: MCS$L_STATUS
access: read only
mechanism: by reference

New status to set. If you omit this argument, the argspec's status is unchanged.

C Binding

```
MCS_STATUS MCS_arglist_setIndexValue (
               MCS_VALUE *argument_list_value,
               MCS_LONGINT index_number,
               MCS_VALUE *argument_value,
               MCS_STATUS argument_status)
```

Description

The *MCS_arglist_setIndexValue* routine finds an argspec by index, frees its old value and status, and sets the argpsec value and status to the ones passed in. Following this call, the argspec in the argument list points to the actual data pointed to by the *argument_value* argument; therefore, do not call *MCS_list_free* or *MCS_datatype_free* on the *argument_value* argument, since to do so would corrupt the argument list.

Condition Values

ERRARGLIST_SETINDX	Error attempting to set the argument list value; see the error stack for the cause.
SUCCESS	Normal successful completion.

See Also

MCS_arglist_setNameValue

MCS_arglist_setNameValue

Modifies the value and status of an argspec containing the specified named argument.

OpenVMS Format

status = MCS$arglist_setNameValue argument_list_value, argument_name, [argument_value,] [argument_status]

Arguments

argument_list_value
type: MCS$R_VALUE
access: read/write
mechanism: by reference

Value structure pointing to argument list containing the argspec to be modified.

argument_name
type: MCS$R_STRINGDSC
access: read only
mechanism: by descriptor

Name of argument to set.

argument_value
type: MCS$R_VALUE
access: read only
mechanism: by reference

Optional. Value structure containing the new value to set. Following the operation, the argspec and this value structure will refer to the same memory. If you omit this argument, the current value structure remains intact. This is useful if you want to set only the status.

argument_status
type: MCS$L_STATUS
access: read only
mechanism: by reference

Optional. New status to set. If you omit this argument, the argspec's status is unchanged.

C Binding

```
MCS_STATUS MCS_arglist_setNameValue (
               MCS_VALUE *argument_list_value,
               MCS_STRING argument_name,
               MCS_VALUE *argument_value,
               MCS_STATUS argument_status)
```

Description

The *MCS_arglist_setNameValue* routine finds an argspec by name, frees its old value and status, and sets the argpsec value and status to the ones passed in. Following this call, the argspec in the argument list points to the actual data pointed to by the *argument_value* argument; therefore, do not call *MCS_list_free* or *MCS_datatype_free* on the *argument_value* argument, since to do so would corrupt the argument list.

Condition Values

ERRARGLIST_SETNAME	Error attempting to set the argument list value; see the error stack for the cause.
SUCCESS	Normal successful completion.

See Also

MCS_arglist_setIndexValue

MCS_check_notices

Returns a list of any notices for the specified element.

OpenVMS Format

```
status = MCS$check_notices   element_elementid, [notice_list_value]
```

Arguments

element_elementid

type: MCS$R_ELEMENTID
access: read only
mechanism: by reference

Element ID of the element to check.

notice_list_value

type: MCS$R_VALUE
access: write only
mechanism: by reference

Optional. Value structure pointing to the list of returned notices. The application is responsible for allocating the value structure and for freeing the resulting list after use with list_free.

C Binding

```
MCS_STATUS MCS_check_notices (
              MCS_ELEMENTID *element_elementid,
              MCS_VALUE *notice_list_value)
```

Description

The *MCS_check_notices* routine returns a value structure containing any notices for the specified element.

Condition Values

If the *notice_list_value* argument is specified:	
ERRCHKMSG	An error occurred while checking messages; check the error stack for the cause.
SUCCESS	Normal successful completion.
If the *notice_list_value* argument is omitted:	
NOTICE	Notices were found on the element.
NONOTICE	No notices were found on the element.
MUSTABORT	Abort transaction; database corrupt.
ERR_NO_ACTIVE_TXN	There is no active transaction.

See Also

MCS_clear_notices
MCS_force_notices
MCS_read_notice

MCS_clear_notices

Removes a notice or a set of notices from an element or set of elements.

OpenVMS Format

```
status = MCS$clear_notices  element_elementid, scope_mask, [notice_list_value]
```

Arguments

element_elementid

type: MCS$R_ELEMENTID
access: read only
mechanism: by reference

Element ID of the element on which to clear notices.

scope_mask

type: MCS$L_LONGINT
access: read only
mechanism: by reference

Longword mask passed by reference telling the repository the scope in which to clear the notice. Its value is the logical OR of the following constants (C binding/OpenVMS binding):

- MCS_NOTICES_CLEAR_LOCAL/MCS$K_NOTICES_CLEAR_LOCAL—Clears the specified notice(s) at the element specified by *element_elementid*.
- MCS_NOTICES_CLEAR_DOWN/MCS$K_NOTICES_CLEAR_DOWN—Clears the specified notice(s) at *element_elementid*'s children recursively.
- MCS_NOTICES_CLEAR_UP/MCS$K_NOTICES_CLEAR_UP—Clears the specified notice(s) at *element_elementid*'s owners recursively.

notice_list_value

type: MCS$R_VALUE
access: read only
mechanism: by reference

Optional. Value structure containing the list of specific notices to be cleared. If you omit this argument, all notices in the specified scope are cleared.

C Binding

```
MCS_STATUS MCS_clear_notices (
               MCS_ELEMENTID *element_elementid,
               MCS_LONGINT scope_mask,
               MCS_VALUE *notice_list_value )
```

Description

The *MCS_clear_notices* routine clears the notices specified in the *notice_list_value* argument. If this argument is not specified, all notices are cleared.

Condition Values

ERRCLRMSG	Error occurred while clearing notices; check the error stack for the cause.
MUSTABORT	Abort transaction; database corrupt.
ERR_NO_ACTIVE_TXN	There is no active transaction.
SUCCESS	Normal successful completion.

See Also

MCS_check_notices
MCS_force_notices
MCS_read_notice

MCS_datatype_compare

Compares the data values contained in two value structures. See the Description section for a list of the types you can compare.

OpenVMS Format

```
status = MCS$datatype_compare   datatype_value_1, datatype_value_2, compare_flag
```

Arguments

datatype_value_1
type: MCS$R_VALUE
access: read only
mechanism: by reference

Value structure containing first value to compare.

datatype_value_2
type: MCS$R_VALUE
access: read only
mechanism: by reference

Value structure containing second value to compare.

compare_flag
type: MCS$L_LONGINT
access: write only
mechanism: by reference

Result of the comparison:

- If compare_flag = 0, the values are equal.
- If compare_flag = 1, value1 is greater than value2.
- If compare_flag = –1, value1 is less than value2.

For some data types, greater than and less than are not meaningful. In these cases a return value of –1 means not equal.

C Binding

```
MCS_STATUS MCS_datatype_compare (
               MCS_VALUE *datatype_value_1,
               MCS_VALUE *datatype_value_2,
               MCS_LONGINT *compare_flag )
```

Description

The *MCS_datatype_compare* routine compares two values of compatible data types and returns the result of the comparison in an argument. The valid data types are:

BOOLEAN
DOUBLE
ELEMENTID
FLOAT
LONGINT
MEMBLOCK
NOTICE
SMALLINT
STRING
STRINGDSC
VMSTIME

The *MCS_datatype_compare* routine makes the necessary conversions to compare compatible values such as different types of numbers. Strings and memory blocks are compared on a byte-by-byte basis.

Condition Values

ILLDATATYPE	Illegal data type for comparison.
INVDATACOMP	Comparison between these two data types is invalid.
SUCCESS	Normal successful completion.

See Also

MCS_elmid_equal
MCS_elmid_isNull

MCS_datatype_copy

Makes a copy of a value structure and the data stored in the value structure. See the Description section for a list of data types that you can copy.

OpenVMS Format

```
status = MCS$datatype_copy   datatype_value_in, datatype_value_out
```

Arguments

datatype_value_in

type: MCS$R_VALUE
access: read only
mechanism: by reference

Value structure to copy.

datatype_value_out

type: MCS$R_VALUE
access: write only
mechanism: by reference

Value structure filled in with a copy of the input value structure. The application is responsible for allocating the value structure. The data in *datatype_value_out* is always a copy of the data in *datatype_value_in*.

C Binding

```
MCS_STATUS MCS_datatype_copy (
              MCS_VALUE *datatype_value_in,
              MCS_VALUE *datatype_value_out )
```

Description

The *MCS_datatype_copy* routine makes a copy of a value structure and the data the value structure points to. The application must call *MCS_datatype_free* to remove any memory associated with *datatype_value_out*.

You can copy the following data types:

BOOLEAN
DOUBLE
ELEMENTID

FLOAT
LIST
LONGINT
MEMBLOCK
NOTICE
SCAN
SMALLINT
STRING
STRINGDSC
VMSTIME

Condition Values

SUCCESS	Normal successful completion.

See Also

MCS_datatype_free
MCS_datatype_new

MCS_datatype_datatype

Returns the name of the data type stored in a value structure.

OpenVMS Format

status = MCS$datatype_datatype datatype_value, datatype_name

Arguments

datatype_value

type: MCS$R_VALUE
access: read only
mechanism: by reference

Value structure.

datatype_name

type: MCS$R_STRINGDSC
access: write only
mechanism: by descriptor

Name of the data type stored in the value structure. The application is responsible for initializing the descriptor and for deallocating the memory after use.

C Binding

```
MCS_STATUS MCS_datatype_datatype (
              MCS_VALUE *datatype_value,
              MCS_STRING *datatype_name )
```

Description

The *MCS_datatype_datatype* routine returns the name of the data type stored in the value structure.

Condition Values

ERR_BAD_DTYPE	An unknown data type is stored in the value structure.
ILLDATATYPE	Illegal data type for this operation.
SUCCESS	Normal successful completion.

MCS_datatype_free

Frees the memory occupied by data pointed to by a value structure.

OpenVMS Format

```
status = MCS$datatype_free   datatype_value
```

Arguments

datatype_value

type: MCS$R_VALUE
access: read/write
mechanism: by reference

Value structure containing the data to free. The value structure itself is not freed.

C Binding

```
MCS_STATUS MCS_datatype_free (
               MCS_VALUE *datatype_value )
```

Description

The *MCS_datatype_free* routine frees any memory under Oracle CDD/Repository control associated with the value structure, but does not free the value structure itself. Memory associated with a value structure is considered under Oracle CDD/Repository control if a copy of the data was made when the value structure was created. This happens, for example, if you call MCS_datatype_ new and specify ISCOPY. Other routines and messages that allocate memory under Oracle CDD/Repository control are as follows:

- *MCS_list_new*
- *MCS_scan_new*
- *MCS_datatype_copy*
- *MCS_scan_insert_with_args* (makes a copy of the argument for storage on the relationship)

- *MCS_dispatch_op* and *MCS_dispatch_superOp* with output arguments
- SETPROP message (makes a copy of the argument for storage in property values)

If memory associated with the value structure is not under Oracle CDD/Repository control, *MCS_datatype_free* does not free that memory. However, the call resets the value structure; it is an error to use any value structure after it has had *MCS_datatype_free* called on it except to reinitialize it with another value.

Routines other than the ones listed in this section (and including *MCS_datatype_new* with NOTCOPY specified) do not copy the data when initializing a value structure. Instead, they copy the value structure itself, including pointers to the data referenced by the value structure. Calling *MCS_datatype_free* on a copied value structure prematurely can therefore corrupt the value structure that was copied.

You can use this routine in place of the *MCS_list_free* and *MCS_scan_free* routines for lists and scans.

When called on a list, *MCS_datatype_free* frees all entries on the specified list. The effect is to call *MCS_datatype_free* recursively on each list entry. If the value of the list entry is another list, *MCS_datatype_free* is called recursively on that entry. This means that *MCS_datatype_free* can be used to free an argument list, which is a list that can contain lists.

When called on a scan, *MCS_datatype_free* releases the resources associated with the scan.

Condition Values

ILLDATATYPE	Illegal data type for this operation.
NOTADATATYPE	Unknown data type name.
SUCCESS	Normal successful completion.

See Also

MCS_list_free
MCS_scan_free

MCS_datatype_length

Returns the length in bytes of the data stored in a value structure, optionally allowing you to specify a data type other than the one stored. See the Description section for a list of data types that are valid with this routine.

OpenVMS Format

status = MCS$datatype_length datatype_value, value_length, [datatype_name]

Arguments

datatype_value
type: MCS$R_VALUE
access: read only
mechanism: by reference

Value structure.

value_length
type: MCS$L_LONGINT
access: write only
mechanism: by reference

Length in bytes of the value.

datatype_name
type: MCS$R_STRINGDSC
access: read only
mechanism: by descriptor

Optional. Name of the data type for which to return the length. If omitted, the routine returns the length for the data type stored in the value structure.

C Binding

```
MCS_STATUS MCS_datatype_length (
               MCS_VALUE *datatype_value,
               MCS_LONGINT *value_length
               MCS_STRING datatype_name )
```

Description

The *MCS_datatype_length* routine returns the length of the data the value structure stores. Use this routine to determine how much memory to allocate before you call *MCS_datatype_read*.

If you specify the optional *datatype_name* value, the routine returns the number of bytes needed to store the data in the format of the data type specified by *datatype_name*.

You can use this routine when *datatype_value* stores one of the following data types:

BOOLEAN
DOUBLE
ELEMENTID
FLOAT
LONGINT
MEMBLOCK
NOTICE
SMALLINT
STRING
STRINGDSC
VMSTIME

Condition Values

ILLDATATYPE	Illegal data type for this operation.
SUCCESS	Normal successful completion.

See Also

MCS_datatype_read

MCS_datatype_new

Creates a value structure containing specified data. See the Description section for a list of data types that can be stored using this call.

OpenVMS Format

```
status = MCS$datatype_new   datatype_value, datatype_name, [datatype_data,] copy_flag
```

Arguments

datatype_value
type: MCS$R_VALUE
access: write only
mechanism: by reference

Value structure to initialize. The application is responsible for allocating the value structure.

datatype_name
type: MCS$R_STRINGDSC
access: read only
mechanism: by descriptor

Name of the data type to store in the new value structure.

datatype_data
type: MCS$R_DATATYPE
access: read only
mechanism: by reference

Optional. Pointer to the value of the data to be stored. If you omit this argument, the value structure is initialized to contain the specified data type but no data is stored in it.

copy_flag
type: MCS$L_BOOLEAN
access: read only
mechanism: by reference

Determines whether *MCS_datatype_new* allocates memory and makes a copy of *datatype_data*. This argument is ignored if *datatype_data* is null. See the Description section for the values of this argument.

C Binding

```
MCS_STATUS MCS_datatype_new (
               MCS_VALUE *datatype_value,
               MCS_STRING datatype_name,
               MCS_DATATYPE *datatype_data,
               MCS_BOOLEAN copy_flag )
```

Description

The *MCS_datatype_new* routine initializes a value structure and optionally stores data in it. The routine is valid for the following data types:

BOOLEAN
DOUBLE
ELEMENTID
FLOAT
LONGINT
MEMBLOCK
NOTICE
SMALLINT
STRING
STRINGDSC
VMSTIME

To create a value structure that contains a list, use the *MCS_list_new* routine. To create a value structure that contains a scan, use the MCS_scan_new routine.

The *copy_flag* argument determines whether the new value structure refers to the actual data passed in or to a copy of the data. The possible values for this argument follow (C binding/OpenVMS binding):

- MCS_DATATYPE_ISCOPY/MCS$K_DATATYPE_ISCOPY—Allocates memory for a copy of the data.
- MCS_DATATYPE_NOTCOPY/MCS$K_DATATYPE_NOTCOPY—Does not allocate memory, but instead points to the actual data.

If you request a copy (by specifying ISCOPY), you are responsible for freeing the memory allocated for the copy by eventually calling *MCS_datatype_free* on the value structure. Conversely, if you do not request a copy, you must be careful not to call *MCS_datatype_free* on the value structure too early, since to do so invalidates other pointers to the data.

Condition Values

ILLDATATYPE	Illegal data type for this operation.
NOTADATATYPE	*datatype_name* does not specify a data type.
SUCCESS	Normal successful completion.

See Also

MCS_datatype_free
MCS_list_new
MCS_scan_new

MCS_datatype_read

Returns the data from a value structure in a form that is useful to the application. See the Description section for a list of data types that are valid with this routine.

OpenVMS Format

status = MCS$datatype_read datatype_value, datatype_name, datatype_data

Arguments

datatype_value
type: MCS$R_VALUE
access: read only
mechanism: by reference

Value structure from which to read the value.

datatype_name
type: MCS$R_STRINGDSC
access: read only
mechanism: by descriptor

Name of the data type in which to return the value.

datatype_data
type: MCS$R_DATATYPE
access: write only
mechanism: by reference

Pointer to a buffer to which *MCS_datatype_read* copies the data value pointed to by the value structure. The buffer must be large enough to hold the value of *datatype_value* when it is formatted in the type *datatype_name*. If the type is STRINGDSC, then this argument is a pointer to a string descriptor, which the user must initialize.

C Binding

```
MCS_STATUS MCS_datatype_read (
               MCS_VALUE *datatype_value,
               MCS_STRING datatype_name,
               MCS_DATATYPE *datatype_data )
```

MCS_datatype_read

Description

The *MCS_datatype_read* routine returns a data value from a value structure in a format useful to the application. The *datatype_name* argument specifies the data type in which to return the data; this can be different from the data type stored in *datatype_value*, as long as the data types are compatible. Oracle CDD/Repository performs the necessary conversions.

datatype_data is the address of a buffer to which *MCS_datatype_read* copies the actual data pointed to by the value structure. The application must allocate the proper storage for the data type (unless the requested data type is a dynamic string descriptor). You can determine the amount of storage to allocate by calling *MCS_datatype_length*, specifying the data type in which you want the data returned.

The following data types are valid:

BOOLEAN
DOUBLE
ELEMENTID
FLOAT
LONGINT
MEMBLOCK
NOTICE
SMALLINT
STRING
STRINGDSC
VMSTIME

You cannot use *MCS_datatype_read* to get a list or scan value. Use *MCS_list_get* and the routines that retrieve scan elements (MCS_scan_get*) to access lists and scans, respectively. Use *MCS_arglist_findArg* and *MCS_arglist_getArg* to read the contents of argument lists.

Condition Values

ILLDATATYPE	Illegal data type for this operation.
INVCVT	Cannot convert between these two data types.
OVERFLOW	Input value was too large for output data type.
SUCCESS	Normal successful completion.

TRUNC	Output value was truncated.
UNDERFLOW	Input value was too precise for output data type.

See Also

MCS_arglist_findArg
MCS_arglist_getArg
MCS_datatype_length
MCS_list_get
MCS_scan_get*

MCS_db_close

Detaches a specified repository from the current session.

OpenVMS Format

```
status = MCS$db_close   session_handle, db_name
```

Arguments

session_handle

type: MCS$R_SESSION
access: read only
mechanism: by reference

Session handle.

db_name

type: MCS$R_STRINGDSC
access: read only
mechanism: by descriptor

Name of a directory containing the repository to be detached from the repository session.

C Binding

```
MCS_STATUS MCS_db_close (
               MCS_SESSION *session_handle,
               MCS_STRING db_name )
```

Description

The *MCS_db_close* routine detaches the specified repository database from the repository session. All element IDs associated with the repository become invalid. No transactions can be open when calling *MCS_db_close*.

Condition Values

SUCCESS	Normal successful completion.

MCS_db_free

Deletes a repository.

OpenVMS Format

status = MCS$db_free session_handle, db_name

Arguments

session_handle
type: MCS$R_SESSION
access: read only
mechanism: by reference

Session handle.

db_name
type: MCS$R_STRINGDSC
access: read only
mechanism: by descriptor

Name of a directory containing a repository.

C Binding

```
MCS_STATUS MCS_db_free (
               MCS_SESSION *session_handle,
               MCS_STRING db_name )
```

MCS_db_free

Description

The *MCS_db_free* routine deletes a repository. Specify the name of the repository as a directory specification. No transactions can be active when you call *MCS_db_free*. If the repository was attached to the session with a call to *MCS_initiate_database*, then it will be detached from the session and element IDs associated with it will become invalid.

Condition Values

SUCCESS	Normal successful completion.

MCS_db_new

Creates a new repository.

OpenVMS Format

```
status = MCS$db_new   session_handle, db_name, anchorroot_name
```

Arguments

session_handle
type: MCS$R_SESSION
access: read only
mechanism: by reference

Session handle.

db_name
type: MCS$R_STRINGDSC
access: read only
mechanism: by descriptor

Name of an empty directory in which to create the new repository. The device associated with the directory must have been mounted through the Distributed File Server (DFS).

anchorroot_name
type: MCS$R_STRINGDSC
access: read only
mechanism: by descriptor

Optional. Name of directory root for the new repository. If specified as 0, Oracle CDD/Repository places the new repository files in subdirectories of the current database anchor. Specify a specific directory root to place the repository file system ([.CONTEXTS...], [.PARTITIONS...], and [.DELTAFILES...]) on a different root than that of the database files.

You must specify this argument if you want to access repository files on a remote node. The device associated with the directory cannot have been mounted through DFS.

You can specify anchorroot_name on a local (host) machine, but not on a remote (client) machine.

MCS_db_new

C Binding

```
MCS_STATUS MCS_db_new (
               MCS_SESSION *session_handle,
               MCS_STRING db_name,
               MCS_STRING anchorroot_name )
```

Description

The *MCS_db_new* routine creates a new repository. Specify the name of the repository as a directory specification; the directory must be empty. No transactions can be active when you call *MCS_db_new*.

Condition Values

SUCCESS	Normal successful completion.

MCS_dispatch_op

Sends a message to an element.

OpenVMS Format

status = MCS$dispatch_op element_elementid, message_name, [argument_list_value]

Arguments

element_elementid
type: MCS$R_ELEMENTID
access: read only
mechanism: by reference

Element ID of element on which to perform the operation.

message_name
type: MCS$R_STRINGDSC
access: read only
mechanism: by descriptor

Name of message to send. You can use the global variables defined by Oracle CDD/Repository to specify this argument: for example, MCS$r_message_ getProp.

argument_list_value
type: MCS$R_VALUE
access: read/write
mechanism: by reference

Optional. Value structure pointing to argument list of input and output arguments. (This argument is required when the message being sent has required arguments.)

C Binding

```
MCS_STATUS MCS_dispatch_op (
               MCS_ELEMENTID *element_elementid,
               MCS_STRING message_name,
               MCS_VALUE *argument_list_value)
```

MCS_dispatch_op

Description

The *MCS_dispatch_op* routine sends a message to an element, resulting in invocation of the method that implements the message for elements of that type.

Condition Values

MUSTABORT	Abort transaction; database corrupt.
ERR_NO_ACTIVE_TXN	There is no active transaction.
SUCCESS	Normal successful completion.

The *MCS_dispatch_op* routine also can return all status codes returned by the methods that it invokes. Any return status other than SUCCESS indicates that a problem occurred. You can examine the error stack to determine the exact nature of the problem.

See Also

MCS_dispatch_superOp

MCS_dispatch_superOp

Sends a message to an element to invoke the method defined by the supertype of a specified type.

OpenVMS Format

```
status = MCS$dispatch_superOp  elementtype_elementid, element_elementid, message_name,
                               [argument_list_value]
```

Arguments

elementtype_elementid
type: MCS$R_ELEMENTID
access: read only
mechanism: by reference

Element ID of the type whose supertype method is to be invoked.

element_elementid
type: MCS$R_ELEMENTID
access: read only
mechanism: by reference

Element ID of the element on which to perform the operation.

message_name
type: MCS$R_STRINGDSC
access: read only
mechanism: by descriptor

Name of message to send. You can use the global variables defined by Oracle CDD/Repository to specify this argument: for example, MCS$r_message_ getProp.

argument_list_value
type: MCS$R_VALUE
access: read/write
mechanism: by reference

Optional. Value structure pointing to the argument list of input and output arguments. (This argument is required when the message being sent has required arguments.)

MCS_dispatch_superOp

C Binding

```
MCS_STATUS MCS_dispatch_superOp (
               MCS_ELEMENTID *elementtype_elementid,
               MCS_ELEMENTID *element_elementid,
               MCS_STRING message_name,
               MCS_VALUE *argument_list_value)
```

Description

The *MCS_dispatch_superOp* routine sends a message to an element but requests that the supertype method of a specified element type be invoked, not the method defined by the type of the target element. Use this routine when a method you are writing refines a supertype's method. At the point in the method code where you want to invoke the supertype's method, call this routine, specifying the type to which your method belongs.

Condition Values

MUSTABORT	Abort transaction; database corrupt.
ERR_NO_ACTIVE_TXN	There is no active transaction.
SUCCESS	Normal successful completion.

The *MCS_dispatch_superOp* routine also can return all status codes returned by the methods that it invokes. Any return status other than SUCCESS indicates that a problem occurred. You can examine the error stack to determine the exact nature of the problem.

See Also

MCS_dispatch_op

MCS_element_getByName

Returns the element ID of the element with the specified name.

OpenVMS Format

status = MCS$element_getByName elementtype_elementid, element_name, element_elementid

Arguments

elementtype_elementid

type: MCS$R_ELEMENTID
access: read only
mechanism: by reference

Element ID of the type of element to return. This argument restricts the search to elements of that type and its subtypes. You can specify the element ID of ELEMENT to search all types; however, the search is faster if you specify the exact type.

element_name

type: MCS$R_STRINGDSC
access: read only
mechanism: by descriptor

Name of element you want. If the name is of a versioned element and you omit branch information, the element ID of the latest version in the main line of descent is returned. If you omit full directory path information, your current default directory is used, unless element_name references metadata. Wildcards are allowed.

element_elementid

type: MCS$R_ELEMENTID
access: read/write
mechanism: by reference

Element ID of the element. The application is responsible for allocating the space for the element ID. The contents of the element ID must be 0 on input.

MCS_element_getByName

C Binding

```
MCS_STATUS MCS_element_getByName (
               MCS_ELEMENTID *elementtype_elementid,
               MCS_STRING element_name,
               MCS_ELEMENTID *element_elementid)
```

Description

The *MCS_element_getByName* routine returns the element ID of the element with the specified name. Specify the exact element type of the element for the fastest search. If you do not know the exact type, specify the element type ELEMENT, which is faster than NAMED_ELEMENT or VERSION.

If the *elementtype_elementid* you supply is for one of the repository metadata types (for example, ELEMENT_TYPE, PROPERTY_TYPE, MESSAGE, or METHOD) *MCS_element_getByName* returns the instance that is currently active. For example, you can find the active version of the ELEMENT_TYPE element named **VERSION** by specifying its name (MCS_elm_version in the C binding, or MCS$r_elm_version in the OpenVMS binding) without branch or directory information. The routine returns the element ID of the instance that currently controls instantiation of VERSION elements, even if this instance is on a branch or is not the latest version.

If you supply a fully qualified path name for a repository metadata type, the routine does not perform this special-case processing and instead does a regular directory lookup.

If you do not specify a version number, *MCS_element_getByName* uses the following default processing rules:

- returns the latest element in any collection under top
- if there is no version in any collection under top, returns the latest element in any partition

See the *Oracle CDD/Repository Architecture Manual* for more information on Oracle CDD/Repository names.

Condition Values

NOSUCHNAME	No element having the specified name exists.
MUSTABORT	Abort transaction; database corrupt.
ERR_NO_ACTIVE_TXN	There is no active transaction.
SUCCESS	Normal successful completion.

See Also

MCS_element_getName

MCS_element_getName

Returns the name of an element.

OpenVMS Format

status = MCS$element_getName element_elementid, element_name

Arguments

element_elementid

type: MCS$R_ELEMENTID
access: read only
mechanism: by reference

Element ID of the element.

element_name

type: MCS$R_STRINGDSC
access: write only
mechanism: by descriptor

Name of the element. The name includes the directory path and any branch and version information. The application is responsible for initializing the descriptor and for deallocating the memory after use.

C Binding

```
MCS_STATUS MCS_element_getName (
               MCS_ELEMENTID *element_elementid,
               MCS_STRING *element_name )
```

Description

The *MCS_element_getName* routine returns the name of the element.

Condition Values

NONAME	The element has no name.
MUSTABORT	Abort transaction; database corrupt.
ERR_NO_ACTIVE_TXN	There is no active transaction.
SUCCESS	Normal successful completion.

See Also

MCS_element_getByName

MCS_element_getSubTypeList

Returns a list of the immediate subtypes of an element type.

OpenVMS Format

```
status = MCS$element_getSubTypeList   element_elementid, subtype_list_value
```

Arguments

element_elementid

type: MCS$R_ELEMENTID
access: read only
mechanism: by reference

Element ID of the element type (this is an instance of ELEMENT_TYPE).

subtype_list_value

type: MCS$R_VALUE
access: write only
mechanism: by reference

Value structure filled in with a list of element IDs of ELEMENT_TYPE elements. The application is responsible for allocating the value structure but does not need to call list_new on the value structure before calling *MCS_element_getSubTypeList*.

C Binding

```
MCS_STATUS MCS_element_getSubTypeList (
              MCS_ELEMENTID *element_elementid,
              MCS_VALUE *subtype_list_value )
```

Description

The *MCS_element_getSubTypeList* routine returns a list of subtypes of an element type. Each entry in the list contains the element ID of an ELEMENT_TYPE element that represents a subtype of the type specified in the call. If the specified type has no subtypes, the list is empty.

Free the list by using *MCS_list_free*.

Condition Values

MUSTABORT	Abort transaction; database corrupt.
ERR_NO_ACTIVE_TXN	There is no active transaction.
SUCCESS	Normal successful completion.

See Also

MCS_element_getSuperTypeList
MCS_element_getType

MCS_element_getSuperTypeList

Returns a list containing the immediate supertype of an element type.

OpenVMS Format

```
status = MCS$element_getSuperTypeList   element_elementid, supertype_list_value
```

Arguments

element_elementid

type: MCS$R_ELEMENTID
access: read only
mechanism: by reference

Element ID of the element type (this is an instance of ELEMENT_TYPE).

supertype_list_value

type: MCS$R_VALUE
access: write only
mechanism: by reference

Value structure filled in with a list of element IDs of ELEMENT_TYPE elements. The application is responsible for allocating the value structure but does not need to call list_new on the value structure before calling *MCS_element_getSuperTypeList*.

C Binding

```
MCS_STATUS MCS_element_getSuperTypeList (
              MCS_ELEMENTID *element_elementid,
              MCS_VALUE *supertype_list_value )
```

Description

The *MCS_element_getSuperTypeList* routine returns a list of the supertypes of an element type. Because Oracle CDD/Repository is currently a single-inheritance system, this list consists of one entry unless *inst* is at the root of the element type hierarchy, in which case the list is empty.

Free the list by using *MCS_list_free*.

Condition Values

MUSTABORT	Abort transaction; database corrupt.
ERR_NO_ACTIVE_TXN	There is no active transaction.
SUCCESS	Normal successful completion.

See Also

MCS_element_getSubTypeList
MCS_element_getType

MCS_element_getType

Returns an element's type.

OpenVMS Format

status = MCS$element_getType element_elementid, elementtype_elementid

Arguments

element_elementid
type: MCS$R_ELEMENTID
access: read only
mechanism: by reference

Element ID of the element.

elementtype_elementid
type: MCS$R_ELEMENTID
access: write only
mechanism: by reference

Element ID of the ELEMENT_TYPE element that represents the element's type.

C Binding

```
MCS_STATUS MCS_element_getType (
              MCS_ELEMENTID *element_elementid,
              MCS_ELEMENTID *elementtype_elementid )
```

Description

The *MCS_element_getType* routine returns the type of an element.

Condition Values

MUSTABORT	Abort transaction; database corrupt.
ERR_NO_ACTIVE_TXN	There is no active transaction.
SUCCESS	Normal successful completion.
NOSUCHNAME	No element with the given name was found.

See Also

MCS_element_getSubTypeList
MCS_element_getSuperTypeList

MCS_elmid_copy

Copies an element ID.

OpenVMS Format

```
status = MCS$elmid_copy   source_elementid, destination_elementid
```

Arguments

source_elementid

type: MCS$R_ELEMENTID
access: read only
mechanism: by reference

Element ID to be copied.

destination_elementid

type: MCS$R_ELEMENTID
access: write only
mechanism: by reference

New (copied) element ID. The application is responsible for allocating space for the new element ID.

C Binding

```
MCS_STATUS MCS_elmid_copy (
               MCS_ELEMENTID *source_elementid,
               MCS_ELEMENTID *destination_elementid )
```

Description

The *MCS_elmid_copy* routine copies an element ID. You can use this routine to replace the contents of one element ID with those of another.

Condition Values

SUCCESS	Normal successful completion.

MCS_elmid_equal

Tests whether two element IDs refer to the same element.

OpenVMS Format

status = MCS$elmid_equal element_elementid_1, element_elementid_2, equal

Arguments

element_elementid_1
type: MCS$R_ELEMENTID
access: read only
mechanism: by reference

Element ID of the first element.

element_elementid_2
type: MCS$R_ELEMENTID
access: read only
mechanism: by reference

Element ID of the second element.

equal
type: MCS$L_BOOLEAN
access: write only
mechanism: by reference

True if the two element IDs reference the same element; false otherwise.

C Binding

```
MCS_STATUS MCS_elmid_equal (
               MCS_ELEMENTID *element_elementid_1,
               MCS_ELEMENTID *element_elementid_2
               MCS_BOOLEAN *equal )
```

MCS_elmid_equal

Description

The *MCS_elmid_equal* routine returns true if the two element IDs point to the same element.

Condition Values

NOTEQUAL	The elements are different.
SUCCESS	Normal successful completion; the elements are the same.

See Also

MCS_elmid_isNull

MCS_elmid_export_persistent

Converts a session-specific element ID to a persistent element identifier that retains its meaning across sessions.

OpenVMS Format

status = MCS$elmid_export_persistent element_elementid, persistentelement_elementid

Arguments

element_elementid
type: MCS$R_ELEMENTID
access: read only
mechanism: by reference

Element ID to be converted.

persistentelement_elementid
type: MCS$R_VALUE
access: write only
mechanism: by reference

Persistent element identifier. The application is responsible for allocating the value structure and for freeing it by using datatype_free after use.

C Binding

```
MCS_STATUS MCS_elmid_export_persistent (
               MCS_ELEMENTID *element_elementid,
               MCS_VALUE *persistentelement_elementid )
```

Description

Element IDs have meaning only in the session in which they are accessed. The *MCS_elmid_export_persistent* routine creates a persistent element identifier that you can use (through *MCS_elmid_importPersistent*) to identify the same element in a later transaction.

Condition Values

SUCCESS	Normal successful completion.

See Also

MCS_elmid_importPersistent

MCS_elmid_getContext

Returns the current context from an element ID.

OpenVMS Format

status = MCS$elmid_getContext element_elementid, context_elementid

Arguments

element_elementid
type: MCS$R_ELEMENTID
access: read only
mechanism: by reference

Element ID from which to take the current context information.

context_elementid
type: MCS$R_ELEMENTID
access: write only
mechanism: by reference

Element ID of the CONTEXT element that represents the current context. A null element ID is returned if there is no current context. The application is responsible for allocating the element ID.

C Binding

```
MCS_STATUS MCS_elmid_getContext (
               MCS_ELEMENTID *element_elementid
               MCS_ELEMENTID *context_elementid )
```

Description

An element ID encodes the current context along with other information. The *MCS_elmid_getContext* routine extracts the current CONTEXT element from an element ID.

Condition Values

MUSTABORT	Abort transaction; database corrupt.
ERR_NO_ACTIVE_TXN	There is no active transaction.
SUCCESS	Normal successful completion.

See Also

MCS_elmid_getPersistentProcess
MCS_elmid_getSession
MCS_elmid_isNull

MCS_elmid_getPersistentProcess

Returns the current persistent process from an element ID.

OpenVMS Format

status = MCS$elmid_getPersistentProcess element_elementid, persistentprocess_elementid

Arguments

element_elementid
type: MCS$R_ELEMENTID
access: read only
mechanism: by reference

Element ID from which to take the current persistent process information.

persistentprocess_elementid
type: MCS$R_ELEMENTID
access: write only
mechanism: by reference

Element ID of the PERSISTENT_PROCESS element that represents the current persistent process. A null element ID is returned if there is no current persistent process. The application is responsible for allocating the element ID.

C Binding

```
MCS_STATUS MCS_elmid_getPersistentProcess (
               MCS_ELEMENTID *element_elementid
               MCS_ELEMENTID *persistentprocess_elementid )
```

Description

An element ID encodes the current persistent process along with other information. The *MCS_elmid_getPersistentProcess* routine extracts the current PERSISTENT_PROCESS element from an element ID.

MCS_elmid_getPersistentProcess

Condition Values

MUSTABORT	Abort transaction; database corrupt.
ERR_NO_ACTIVE_TXN	There is no active transaction.
SUCCESS	Normal successful completion.

See Also

MCS_elmid_getContext
MCS_elmid_getSession
MCS_elmid_isNull

MCS_elmid_getSession

Returns the current session from an element ID.

OpenVMS Format

```
status = MCS$elmid_getSession   element_elementid, session_handle
```

Arguments

element_elementid

type: MCS$R_ELEMENTID
access: read only
mechanism: by reference

Element from which to take the current session information.

session_handle

type: MCS$R_SESSION
access: write only
mechanism: by reference

Session handle of the current session. The application is responsible for allocating space for the session handle.

C Binding

```
MCS_STATUS MCS_elmid_getSession (
               MCS_ELEMENTID *element_elementid
               MCS_SESSION *session_handle )
```

Description

An element ID encodes the current session along with other information. The *MCS_elmid_getSession* routine extracts the current session from an element ID and returns the session handle of the session.

Condition Values

SUCCESS	Normal successful completion.

See Also

MCS_elmid_getContext
MCS_elmid_getPersistentProcess

MCS_elmid_import_persistent

Converts a persistent element identifier (one that retains its meaning across sessions) to a session-specific element ID.

OpenVMS Format

```
status =MCS$elmid_import_persistent  persistentelement_elementid, session_handle,
                                     element_elementid
```

Arguments

persistentelement_elementid
type: MCS$R_VALUE
access: read only
mechanism: by reference

Persistent element identifier to be converted.

session_handle
type: MCS$R_SESSION
access: read only
mechanism: by reference

Session handle for the session to be associated with the element ID.

element_elementid
type: MCS$R_ELEMENTID
access: write only
mechanism: by reference

Session-specific element ID. The application is responsible for allocating the element ID.

C Binding

```
MCS_STATUS MCS_elmid_import_persistent (
                MCS_VALUE *persistentelement_elementid,
                MCS_SESSION *session_handle,
                MCS_ELEMENTID *element_elementid )
```

MCS_elmid_import_persistent

Description

Element IDs have meaning only in the session in which they are accessed. The *MCS_elmid_export_persistent* routine can create a persistent element identifier that continues to identify the same element from one session to the next. However, you cannot use this element identifier in place of an element ID in Oracle CDD/Repository callable interface calls. This routine converts a persistent element identifier to a session-specific element ID.

Condition Values

SUCCESS	Normal successful completion.

See Also

MCS_elmid_export_persistent

MCS_elmid_isNull

Tests whether an element ID is null.

OpenVMS Format

status = MCS$elmid_isNull element_elementid, isnull_flag

Arguments

element_elementid

type: MCS$R_ELEMENTID
access: read only
mechanism: by reference

Element ID to test.

isnull_flag

type: MCS$L_BOOLEAN
access: write only
mechanism: by reference

True if the element ID is null.

C Binding

```
MCS_STATUS MCS_elmid_isNull (
               MCS_ELEMENTID *element_elementid,
               MCS_BOOLEAN *isnull_flag)
```

Description

The *MCS_elmid_isNull* routine tests whether an element ID has a null value. An element ID can have a null value if it is the value of a property that has not been set.

MCS_elmid_isNull

Condition Values

NOTNULL	The element ID is not null.
SUCCESS	Normal successful completion; the element ID is null.

See Also

MCS_elmid_equal

MCS_elmid_isSubtype

Tests whether one element type is a subtype of another.

OpenVMS Format

status = MCS$elmid_isSubtype subtype_elementid, supertype_elementid, issubtype_flag

Arguments

subtype_elementid
type: MCS$R_ELEMENTID
access: read only
mechanism: by reference

Element ID of the suspected subtype.

supertype_elementid
type: MCS$R_ELEMENTID
access: read only
mechanism: by reference

Element ID of the suspected supertype.

issubtype_flag
type: MCS$L_BOOLEAN
access: write only
mechanism: by reference

True if the suspected subtype is a subtype of the suspected supertype.

C Binding

```
MCS_STATUS MCS_elmid_isSubtype (
               MCS_ELEMENTID *subtype_elementid,
               MCS_ELEMENTID *supertype_elementid,
               MCS_BOOLEAN *issubtype_flag)
```

MCS_elmid_isSubtype

Description

The *MCS_elmid_isSubtype* routine tests if one element type is a subtype of the other. If the two types are the same, they are considered to be subtypes of one another, since an element type is a subtype of itself.

Condition Values

NOTSUBTYPE	The suspected subtype is not a subtype of the suspected supertype.
MUSTABORT	Abort transaction; database corrupt.
ERR_NO_ACTIVE_TXN	There is no active transaction.
SUCCESS	Normal successful completion; the suspected subtype is a subtype of the suspected supertype.

MCS_errorstack_clear

Removes a specified entry from the error stack and frees the space it occupied.

OpenVMS Format

```
status = MCS$errorstack_clear   index_number
```

Arguments

index_number

type: MCS$L_LONGINT
access: read only
mechanism: by reference

Index number of the entry to be removed. Zero is the top of the stack.

C Binding

```
MCS_STATUS mcs_errorstack_clear (
               MCS_LONGINT index_number)
```

Description

The *MCS_errorstack_clear* routine removes a specified entry from the error stack and releases the space it occupied. The entry on top of the error stack is number 0; entries below it are numbered in increasing order.

Condition Values

ERRSTACK_CLEAR	An error occurred while attempting to clear the entry; check the error stack for the cause.
INDEXTOOLARGE	The index is too large for the current size of the error stack.
SUCCESS	Normal successful completion.

MCS_errorstack_clear

See Also

MCS_errorstack_clearAll

MCS_errorstack_clearAll

Removes all entries from the error stack and frees the space they occupied.

OpenVMS Format

```
status = MCS$errorstack_clearAll
```

Arguments

None

C Binding

```
MCS_STATUS MCS_errorstack_clearAll ()
```

Description

The *MCS_errorstack_clearAll* routine removes all entries from the error stack and frees the space they occupied.

Condition Values

ERRSTACK_CLEARALL	An error occurred while attempting to clear the error stack; check the error stack for the cause.
SUCCESS	Normal successful completion.

See Also

MCS_errorstack_clear

MCS_errorstack_format

Formats an error message from an error stack entry.

OpenVMS Format

```
status = MCS$errorstack_format   index_number, message_text
```

Arguments

index_number

type: MCS$L_LONGINT
access: read only
mechanism: by reference

Index number of the entry to be formatted. Zero is the top of the stack.

message_text

type: MCS$R_STRINGDSC
access: write only
mechanism: by descriptor

Formatted message. The application is responsible for initializing the descriptor and deallocating the memory after use.

C Binding

```
MCS_STATUS MCS_errorstack_format (
               MCS_LONGINT index_number,
               MCS_STRING *message_text)
```

Description

The *MCS_errorstack_format* routine obtains an error message (by index number) from the error stack, formats it, and places it in a buffer.

Condition Values

ERRSTACK_FORMAT	An error occurred while attempting to format the message; check the error stack for the cause.
INDEXTOOLARGE	The index was too large for the current size of the error stack.
SUCCESS	Normal successful completion.

See Also

MCS_errorstack_getStatus

MCS_errorstack_getCurrentSize

Returns the number of entries currently on the error stack.

OpenVMS Format

status = MCS$errorstack_getCurrentSize current_size

Arguments

current_size

type: MCS$L_LONGINT
access: write only
mechanism: by reference

Current error stack size.

C Binding

```
MCS_STATUS MCS_errorstack_getCurrentSize (
               MCS_LONGINT *current_size)
```

Description

The *MCS_errorstack_getCurrentSize* routine returns the number of entries currently on the error stack.

Condition Values

ERRSTACK_GETCURSIZE	An error occurred while attempting to get the size of the error stack; check the error stack for the cause.
SUCCESS	Normal successful completion.

MCS_errorstack_getMaxSize

Returns the maximum number of entries that the error stack may contain.

OpenVMS Format

```
status = MCS$errorstack_getMaxSize   maximum_size
```

Arguments

maximum_size

type: MCS$L_LONGINT
access: write only
mechanism: by reference

Maximum number of entries the error stack may contain.

C Binding

```
MCS_STATUS MCS_errorstack_getMaxSize (
               MCS_LONGINT *maximum_size)
```

Description

The *MCS_errorstack_getMaxSize* routine returns the maximum number of entries that the error stack may contain. You can increase the maximum size of the error stack by calling *MCS_errorstack_setMaxSize*.

Condition Values

ERRSTACK_GETMAXSIZE	An error occurred while attempting to get the error stack maximum size; check the error stack for the cause.
SUCCESS	Normal successful completion.

See Also

MCS_errorstack_setMaxSize

MCS_errorstack_getStatus

Returns the status code for an error stack entry.

OpenVMS Format

status = MCS$errorstack_getStatus index_number, status_code

Arguments

index_number

type: MCS$L_LONGINT
access: read only
mechanism: by reference

Index number of the entry to be read. Zero is the top of the stack.

status_code

type: MCS$L_STATUS
access: write only
mechanism: by reference

Status code for the specified error stack entry.

C Binding

```
MCS_STATUS MCS_errorstack_getStatus (
               MCS_LONGINT index_number,
               MCS_STATUS *status_code)
```

Description

The *MCS_errorstack_getStatus* routine returns the status code for the specified error stack entry.

Condition Values

ERRSTACK_GETSTATUS	An error occurred while attempting to get the error status; check the error stack for the cause.

INDEXTOOLARGE	The index was too large for the current size of the error stack.
SUCCESS	Normal successful completion.

See Also

MCS_errorstack_format

MCS_errorstack_set

Adds a new entry to the top of the error stack.

OpenVMS Format

status = MCS$errorstack_set status_code, [argument_count,] [argument_list]

Arguments

status_code
type: MCS$L_STATUS
access: read only
mechanism: by reference

Status code for the new entry.

argument_count
type: MCS$L_LONGINT
access: read only
mechanism: by reference

Optional. Number of arguments in the argument list for the new entry. You can omit this argument if there are no arguments for the entry.

argument_list
type: MCS$R_VALUE
access: read only
mechanism: by reference

Optional. Argument list for the new entry. You can omit this argument if there are no arguments for the entry.

C Binding

```
MCS_STATUS MCS_errorstack_set (
               MCS_STATUS status_code,
               MCS_LONGINT argument_count,
               MCS_VALUE *argument_list)
```

Description

The *MCS_errorstack_set* routine pushes a new entry onto the error stack. An error-stack entry consists of the following:

- the status code
- the argument count
- the list of arguments

The argument count specifies the number of arguments; the list of arguments is a list of argument specifiers used in the formatting of an error message from the error-stack entry by the *MCS_errorstack_format* routine. The number of arguments actually used is the minimum of the *argument_count* value and the size of the list.

If the error stack is full, you can extend it by calling *MCS_errorstack_getMaxSize* to determine the current maximum size, then *MCS_errorstack_setMaxSize* to increase it.

Condition Values

ERRSTACK_SET	An error occurred while attempting to push an entry on the error stack; check the error stack for the cause.
STACKFULL	The error stack is full.
SUCCESS	Normal successful completion.

See Also

MCS_errorstack_format
MCS_errorstack_setMaxSize

MCS_errorstack_setMaxSize

Sets the maximum number of error stack entries.

OpenVMS Format

```
status = MCS$errorstack_setMaxSize   maximum_size
```

Arguments

maximum_size

type: MCS$L_LONGINT
access: read only
mechanism: by reference

New maximum number of entries in the error stack.

C Binding

```
MCS_STATUS MCS_errorstack_setMaxSize (
               MCS_LONGINT maximum_size)
```

Description

The *MCS_errorstack_setMaxSize* routine sets the maximum number of error-stack entries. At Oracle CDD/Repository startup, the default size is set to MCS$K_MSGVEC_LEN. If the stack reaches the maximum size, you cannot use *MCS_errorstack_set* to introduce any more entries unless you increase the maximum size.

Condition Values

ERRSTACK_SETMAXSIZE	An error occurred while attempting to increase the error stack maximum size; check the error stack for the cause.
SUCCESS	Normal successful completion.

See Also

MCS_errorstack_getMaxSize

MCS_fileop_copy

Performs a journaled file copy operation.

OpenVMS Format

```
status = MCS$fileop_copy   valid_elementid, input_name, output_name
```

Arguments

valid_elementid

type: MCS$R_ELEMENTID
access: read only
mechanism: by reference

Element ID of any element being used in the session.

input_name

type: MCS$R_STRINGDSC
access: read only
mechanism: by reference

Name of the input file.

output_name

type: MCS$R_STRINGDSC
access: read only
mechanism: by reference

Name of the output file.

C Binding

```
MCS_STATUS MCS_fileop_copy(
                MCS_ELEMENTID *valid_elementid,
                MCS_STRING input_name,
                MCS_STRING output_name)
```

Description

The *MCS_fileop_copy* routine copies a file and adds an entry to the fileop journal file. If the transaction in which the operation occurs is aborted or the system fails, the output file is automatically deleted.

Condition Values

BADJNLWRT	The journal write operation failed.
BADMALLOC	Memory allocation failed.
FILEIOERR	I/O error.
FILEISDIR	Input file is a directory.
SUCCESS	Normal successful completion.

See Also

MCS_fileop_rename

MCS_fileop_delete

Performs a journaled file delete operation.

OpenVMS Format

status = MCS$fileop_delete valid_elementid, name

Arguments

valid_elementid

type: MCS$R_ELEMENTID
access: read only
mechanism: by reference

Element ID of any element being used in the session.

name

type: MCS$R_STRINGDSC
access: read only
mechanism: by reference

Name of the file to be deleted.

C Binding

```
MCS_STATUS MCS_fileop_delete(
               MCS_ELEMENTID *valid_elementid,
               MCS_STRING name)
```

Description

The *MCS_fileop_delete* routine deletes a file and adds an entry to the fileop journal file. If the transaction in which the operation occurs is aborted or the system fails, the file deletion is rolled back.

Condition Values

BADJNLWRT	The journal write operation failed.
BADMALLOC	Memory allocation failed.
BADRENAME	The file rename operation failed.
SUCCESS	Normal successful completion.

See Also

MCS_fileop_journal_create

MCS_fileop_journal_create

Writes a journal record for a file create operation.

OpenVMS Format

status = MCS$fileop_journal_create valid_elementid, name

Arguments

valid_elementid

type: MCS$R_ELEMENTID
access: read only
mechanism: by reference

Element ID of any element being used in the session.

name

type: MCS$R_STRINGDSC
access: read only
mechanism: by reference

Name of the file to be created.

C Binding

```
MCS_STATUS MCS_fileop_journal_create(
               MCS_ELEMENTID *valid_elementid,
               MCS_STRING name)
```

Description

The *MCS_fileop_journal_create* routine allows you to journal arbitrary file creations. Note that the routine does not actually create the file. It only records the fact that the file was created. You must call this routine before creating the file. If the transaction in which the operation occurs is aborted or the system fails, the created file is automatically deleted.

Condition Values

MCS_SUCCESS	Normal successful completion.
MCS_BADMALLOC	Memory allocation failed.
MCS_BADJNLWRT	The journal write operation failed.

See Also

MCS_fileop_delete
MCS_fileop_journal_modify

MCS_fileop_journal_modify

Writes a journal record for a file modification operation.

OpenVMS Format

status = MCS$fileop_journal_modify valid_elementid, name

Arguments

valid_elementid

type: MCS$R_ELEMENTID
access: read only
mechanism: by reference

Element ID of any element being used in the session.

name

type: MCS$R_STRINGDSC
access: read only
mechanism: by reference

Name of the file to be modified.

C Binding

```
MCS_STATUS MCS_fileop_journal_modify(
               MCS_ELEMENTID *valid_elementid,
               MCS_STRING name)
```

Description

The *MCS_fileop_journal_modify* routine allows you to journal a modification to a file. The routine does not actually modify the file. It saves the initial contents of the file and records the fact that the file was modified. You must call this routine before modifying the file. If the transaction in which the operation occurs is aborted or the system fails, the file is automatically restored to its unmodified state.

Condition Values

BADMALLOC	Memory allocation failed.
BADJNLWRT	The journal write operation failed.
FILEIOERR	I/O error.
FILEISDIR	The file to be modified is a directory.
SUCCESS	Normal successful completion.

See Also

MCS_fileop_journal_copy

MCS_fileop_mkdir

Performs a journaled directory creation.

OpenVMS Format

```
status = MCS$fileop_mkdir  valid_elementid, name
```

Arguments

valid_elementid

type: MCS$R_ELEMENTID
access: read only
mechanism: by reference

Element ID of any element being used in the session.

name

type: MCS$R_STRINGDSC
access: read only
mechanism: by reference

Name of the directory to be created.

C Binding

```
MCS_STATUS MCS_fileop_mkdir(
               MCS_ELEMENTID *valid_elementid,
               MCS_STRING name)
```

Description

The *MCS_fileop_mkdir* routine creates a directory and adds an entry to the fileop journal file. If the transaction in which the operation occurs is aborted or the system fails, the directory is automatically deleted.

Condition Values

BADJNLWRT	The journal write operation failed.
BADMALLOC	Memory allocation failed.
BADMKDIR	Directory creation failed.
SUCCESS	Normal successful completion.

See Also

MCS_fileop_rmdir

MCS_fileop_rename

Performs a journaled file rename operation.

OpenVMS Format

status = MCS$fileop_rename valid_elementid, orig, new

Arguments

valid_elementid
type: MCS$R_ELEMENTID
access: read only
mechanism: by reference

Element ID of any element being used in the session.

orig
type: MCS$R_STRINGDSC
access: read only
mechanism: by reference

Original file name.

new
type: MCS$R_STRINGDSC
access: read only
mechanism: by reference

New file name.

C Binding

```
MCS_STATUS MCS_fileop_rename(
               MCS_ELEMENTID *valid_elementid,
               MCS_STRING orig,
               MCS_STRING new)
```

Description

The *MCS_fileop_rename* routine renames a file and adds an entry to the fileop journal file. If the transaction in which the operation occurs is aborted or the system fails, the file is automatically named back to its original name.

Condition Values

BADJNLWRT	The journal write operation failed.
BADMALLOC	Memory allocation failed.
BADRENAME	The rename operation failed.
SUCCESS	Normal successful completion.

See Also

MCS_fileop_copy

MCS_fileop_rmdir

Performs a journaled directory deletion.

OpenVMS Format

status = MCS$fileop_rmdir valid_elementid, name

Arguments

valid_elementid

type: MCS$R_ELEMENTID
access: read only
mechanism: by reference

Element ID of any element being used in the session.

name

type: MCS$R_STRINGDSC
access: read only
mechanism: by reference

Name of the directory to be deleted.

C Binding

```
MCS_STATUS MCS_fileop_rmdir(
               MCS_ELEMENTID *valid_elementid,
               MCS_STRING name)
```

Description

The *MCS_fileop_rmdir* routine deletes a directory and adds an entry to the fileop journal file. If the transaction in which the operation occurs is aborted or the system fails, the directory is automatically restored.

Condition Values

BADJNLWRT	The journal write operation failed.
BADMALLOC	Memory allocation failed.
BADRMDIR	The directory deletion failed.
SUCCESS	Normal successful completion.

See Also

MCS_fileop_mkdir

MCS_fileop_rmlink

Performs a journaled remove link operation.

OpenVMS Format

status = MCS$fileop_rmlink valid_elementid, name

Arguments

valid_elementid

type: MCS$R_ELEMENTID
access: read only
mechanism: by reference

Element ID of any element being used in the session.

name

type: MCS$R_STRINGDSC
access: read only
mechanism: by reference

Name of the link.

C Binding

```
MCS_STATUS MCS_fileop_rmlink(
               MCS_ELEMENTID *valid_elementid,
               MCS_STRING name)
```

Description

The *MCS_fileop_rmlink* routine deletes a symbolic link to a file and adds an entry to the fileop journal file. If the transaction in which the operation occurs is aborted or the system fails, the symbolic link is automatically restored.

Condition Values

BADJNLWRT	The journal write operation failed.
BADMALLOC	Memory allocation failed.
BADRENAME	The rename operation failed.
SUCCESS	Normal successful completion.

See Also

MCS_fileop_symlink

MCS_fileop_symlink

Performs a journaled symbolic link creation.

OpenVMS Format

status = MCS$fileop_symlink valid_elementid, file_name, link

Arguments

valid_elementid

type: MCS$R_ELEMENTID
access: read only
mechanism: by reference

Element ID of any element being used in the session.

file_name

type: MCS$R_STRINGDSC
access: read only
mechanism: by reference

Name of the existing file.

link

type: MCS$R_STRINGDSC
access: read only
mechanism: by reference

Link name.

C Binding

```
MCS_STATUS MCS_fileop_symlink(
                MCS_ELEMENTID *valid_elementid,
                MCS_STRING file_name,
                MCS_STRING link)
```

Description

The *MCS_fileop_symlink* routine creates a symbolic link to a file and adds an entry to the fileop journal file. If the transaction in which the operation occurs is aborted or the system fails, the symbolic link is automatically removed.

Condition Values

BADJNLWRT	The journal write operation failed.
BADSYMLNK	The symbolic link creation operation failed.
SUCCESS	Normal successful completion.

See Also

MCS_fileop_rmlink

MCS_fileop_unjournal_create

Deletes the journal record for the most recent journaled file operation.

OpenVMS Format

```
status = MCS$fileop_unjournal_create   valid_elementid
```

Arguments

valid_elementid

type: MCS$R_ELEMENTID
access: read only
mechanism: by reference

Element ID of any element being used in the session.

C Binding

```
MCS_STATUS MCS_fileop_unjournal_create(
                MCS_ELEMENTID *valid_elementid)
```

Description

The *MCS_fileop_unjournal_create* routine removes the most recent entry from the fileop journal file. Call this routine if you called *MCS_fileop_journal_create* and the file operation failed.

Condition Values

SUCCESS	Normal successful completion.

See Also

MCS_fileop_journal_create
MCS_fileop_journal_modify

MCS_force_notices

Causes an element to notify owners as if it had been changed.

OpenVMS Format

```
status = MCS$force_notices  element_elementid, cause_flag
```

Arguments

element_elementid

type: MCS$R_ELEMENTID
access: read only
mechanism: by reference

Element ID of element that will notify owners.

cause_flag

type: MCS$L_LONGINT
access: read only
mechanism: by reference

Type of notification to send.

The values for the *cause_flag* argument are as follows (C binding/OpenVMS binding):

- MCS_NOTICES_FORCE_ERASE/MCS$K_NOTICES_FORCE_ERASE—Element has been erased.
- MCS_NOTICES_FORCE_NOTIFY/MCS$K_NOTICES_FORCE_MODIFY—Element has been modified.

C Binding

```
MCS_STATUS MCS_force_notices (
              MCS_ELEMENTID *element_elementid,
              MCS_LONGINT cause_flag )
```

MCS_force_notices

Description

The *MCS_force_notices* routine causes an element to notify its owners as if it had been changed. Use this routine to simulate change in an element when no change actually takes place. For example, when an element represents an object that is stored outside the repository, and that object is changed, you can call *MCS_force_notices* to notify owners of the change even though the element itself does not change.

The *cause_flag* argument specifies what kind of (simulated) change has occurred.

Condition Values

ERRFRCMSG	An error occurred while attempting to force notices; check the error stack for the cause.
MUSTABORT	Abort transaction; database corrupt.
ERR_NO_ACTIVE_TXN	There is no active transaction.
SUCCESS	Normal successful completion.

See Also

MCS_check_notices
MCS_clear_notices
MCS_read_notice

MCS_initiate_database

Establishes access to a repository during a transaction and sets the defaults.

OpenVMS Format

```
status = MCS$initiate_database   database_name, transaction_handle, elementtype_elementid
```

Arguments

database_name

type: MCS$R_STRINGDSC
access: read only
mechanism: by descriptor

Name of the database.

transaction_handle

type: MCS$R_TRANSACTION
access: read only
mechanism: by reference

Handle of the transaction in which to open the repository.

elementtype_elementid

type: MCS$R_ELEMENTID
access: write only
mechanism: by reference

Element ID of the ELEMENT_TYPE element **ELEMENT_TYPE**. The application is responsible for allocating the element ID.

C Binding

```
MCS_STATUS MCS_initiate_database (
               MCS_STRING database_name,
               MCS_TRANSACTION *transaction_handle,
               MCS_ELEMENTID *elementtype_elementid )
```

MCS_initiate_database

Description

The *MCS_initiate_database* routine opens the repository, specifying the transaction handle from the current transaction. This allows information about the current repository to be stored in the session block.

The *elementtype_elementid* argument receives the element ID of the ELEMENT_TYPE element named **ELEMENT_TYPE**. The **instances** property of this element contains all the instances of ELEMENT_TYPE, and the **instances** property of each type contains all instances of that type. By following the **instances** property, you can find all element types; by following the **instances** property again from a particular type, you can find all instances of that type.

Use *MCS_initiate_database* with the following routines:

- *MCS_session_initiate*—To start a session
- *MCS_session_transaction_init*—To start a transaction
- *MCS_session_transaction_term*—To end a transaction
- *MCS_session_terminate*—To end a session

Condition Values

ERRINITIATE	An error occurred while attempting to initiate the database; check the error stack for the cause.
SUCCESS	Normal successful completion.

See Also

MCS_session_terminate

MCS_list_free

Frees a list and all list entries, including value memory under Oracle CDD/Repository control. This routine is equivalent to *MCS_datatype_free*.

OpenVMS Format

```
status = MCS$list_free   list_value
```

Arguments

list_value

type:	MCS$R_VALUE
access:	read/write
mechanism:	by reference

Value structure pointing to the start of the list to free.

C Binding

```
MCS_STATUS MCS_list_free (
               MCS_value *list_value)
```

Description

See the Description section of *MCS_datatype_free* for information about this routine.

Condition Values

SUCCESS	Normal successful completion.

See Also

MCS_datatype_free
MCS_list_new
MCS_list_set

MCS_list_get

Returns a list entry, given an index.

OpenVMS Format

status = MCS$list_get list_value, index_number, data_value

Arguments

list_value

type: MCS$R_VALUE
access: read only
mechanism: by reference

Value structure that contains the list.

index_number

type: MCS$L_LONGINT
access: read only
mechanism: by reference

Zero-based index of the list entry to be retrieved; a value of MCS$K_LIST_END retrieves the last entry.

data_value

type: MCS$R_VALUE
access: write only
mechanism: by reference

Value structure filled in with a copy of the value structure occupying the list entry indicated by the index. The application is responsible for allocating the value structure.

C Binding

```
MCS_STATUS MCS_list_get (
               MCS_VALUE *list_value,
               MCS_LONGINT index_number,
               MCS_VALUE *data_value
               )
```

Description

The *MCS_list_get* routine retrieves a list entry and stores it in a value structure. The routine never copies data pointed to by the list entry, so the list entry and the value structure will point to the same memory following the operation.

Use the constant value MCS_LIST_END (MCS$K_LIST_END) as the index value to access the last entry in the list.

Condition Values

INDEXTOOLARGE	The index was too large for the current size of the list.
SUCCESS	Normal successful completion.

See Also

MCS_list_insert
MCS_list_set

MCS_list_getSize

Returns the number of list entries.

OpenVMS Format

status = MCS$list_getSize list_value, current_size

Arguments

list_value

type: MCS$R_VALUE
access: read only
mechanism: by reference

Value structure pointing to the start of the list.

current_size

type: MCS$L_LONGINT
access: write only
mechanism: by reference

Number of list entries.

C Binding

```
MCS_STATUS MCS_list_getSize (
               MCS_VALUE *list_value,
               MCS_LONGINT *current_size
               )
```

Description

The *MCS_list_getSize* routine returns the number of entries in a list. This number changes as entries are added to and removed from the list.

Condition Values

SUCCESS	Normal successful completion.

MCS_list_insert

Inserts a new entry into a list.

OpenVMS Format

status = MCS$list_insert list_value, index_number, data_value

Arguments

list_value
type: MCS$R_VALUE
access: read/write
mechanism: by reference

List into which the entry is to be inserted.

index_number
type: MCS$L_LONGINT
access: read only
mechanism: by reference

Zero-based index of the list entry before which to insert the new entry. A value of MCS$K_LIST_END places the new entry at the end of the list.

data_value
type: MCS$R_VALUE
access: read only
mechanism: by reference

Value structure to be inserted into the list.

C Binding

```
MCS_STATUS MCS_list_insert (
               MCS_VALUE *list_value,
               MCS_LONGINT index_number,
               MCS_VALUE *data_value
               )
```

MCS_list_insert

Description

The *MCS_list_insert* routine inserts a new entry into a list, increasing the size of the list by one entry. Unless the entry is at the end of the list, existing entries following the new entry are moved out by one position. The routine never copies the actual data pointed to by the *data_value* argument, so the new list entry and the *data_value* value structure point to the same memory following the operation.

Use the constant value MCS_LIST_END (MCS$K_LIST_END) as the index value to insert at the end of the list.

Condition Values

INDEXTOOLARGE	The index was too large for the current size of the list.
SUCCESS	Normal successful completion.

See Also

MCS_list_get
MCS_list_remove
MCS_list_set

MCS_list_new

Creates a list and initializes a value structure to point to it.

OpenVMS Format

```
status = MCS$list_new   list_value, initial_size, size_increment
```

Arguments

list_value

type: MCS$R_VALUE
access: write only
mechanism: by reference

Value structure filled in to contain the new list. The application is responsible for allocating the value structure.

initial_size

type: MCS$L_LONGINT
access: read only
mechanism: by reference

Initial number of list entries.

size_increment

type: MCS$L_LONGINT
access: read only
mechanism: by reference

Number of entries to add each time the list is expanded.

C Binding

```
MCS_STATUS MCS_list_new (
               MCS_VALUE *list_value,
               MCS_LONGINT initial_size,
               MCS_LONGINT size_increment)
```

MCS_list_new

Description

The *MCS_list_new* routine creates an empty list. Oracle CDD/Repository allocates the memory for the list; therefore, the memory containing the list is under Oracle CDD/Repository control, and will be freed by a call to *MCS_list_free* or *MCS_datatype_free*.

You can preallocate the number of list entries you want with the *initial_size* argument. It is much faster to insert new entries into a list if the entries have been preallocated. This argument does not affect the current size of the new list; a new list always has a size of zero entries, and grows by one each time a new entry is inserted.

Condition Values

SUCCESS	Normal successful completion.

See Also

MCS_list_free

MCS_list_remove

Removes and frees a list entry.

OpenVMS Format

status = MCS$list_remove list_value, index_number

Arguments

list_value

type: MCS$R_VALUE
access: read/write
mechanism: by reference

Value structure containing the list from which to remove an entry.

index_number

type: MCS$L_LONGINT
access: read only
mechanism: by reference

Zero-based index of the entry to remove. A value of MCS$K_LIST_END removes the last entry from the list.

C Binding

```
MCS_STATUS MCS_list_remove (
               MCS_VALUE *list_value,
               MCS_LONGINT index_number
               )
```

Description

The *MCS_list_remove* routine removes a list entry and frees all memory (controlled by Oracle CDD/Repository) associated with the entry. This call has the following effect:

1. Calls *MCS_datatype_free* on the entry (or *MCS_list_free*, if the entry is a list), freeing any memory associated with the existing list entry and controlled by Oracle CDD/Repository.
2. Adjusts the current size and structure of the list.

See the Description section of *MCS_datatype_free* for information about how memory referenced by the value structure is treated.

Condition Values

INDEXTOOLARGE	The index was too large for the current size of the list.
SUCCESS	Normal successful completion.

See Also

MCS_list_insert
MCS_list_set

MCS_list_set

Sets the value of an existing list entry.

OpenVMS Format

status = MCS$list_set list_value, index_number, data_value

Arguments

list_value
type: MCS$R_VALUE
access: read/write
mechanism: by reference

Value structure containing the list.

index_number
type: MCS$L_LONGINT
access: read only
mechanism: by reference

Zero-based index of the member whose value is to be changed. An index of MCS$K_LIST_END sets the last list entry.

data_value
type: MCS$R_VALUE
access: read only
mechanism: by reference

Value structure to replace the list entry.

C Binding

```
MCS_STATUS MCS_list_set (
               MCS_VALUE *list_value,
               MCS_LONGINT index_number,
               MCS_VALUE *data_value
               )
```

MCS_list_set

Description

The *MCS_list_set* routine sets the value of an existing list entry. This call has the following effect:

1. Calls *MCS_list_remove* on the list entry, freeing any memory associated with the existing list entry and controlled by Oracle CDD/Repository. (See the description of *MCS_list_remove* for information about how this routine treats memory.)
2. Inserts the new entry at the same place in the list.

The routine never copies data pointed to by the *data_value* value structure. Following the operation, the new list entry and *data_value* point to the same data.

Condition Values

INDEXTOOLARGE	The index was too large for the current size of the list.
SUCCESS	Normal successful completion.

See Also

MCS_list_insert
MCS_list_remove

MCS_read_notice

Reads information from a notice.

OpenVMS Format

```
status = MCS$read_notice   notice_value, sender_elementid, cause_flag
```

Arguments

notice_value

type: MCS$R_VALUE
access: read only
mechanism: by reference

Value structure containing a notice; one of a list of notices returned by MCS_check_notices.

sender_elementid

type: MCS$R_ELEMENTID
access: write only
mechanism: by reference

Element ID of the element that sent the notice.

cause_flag

type: MCS$L_LONGINT
access: write only
mechanism: by reference

Reason the notice was sent. The values for the *cause_flag* argument are the following:

C Binding	OpenVMS Binding	Reason
MCS_NOTICES_ POSSIBLY_INVALID	MCS$K_NOTICES_ POSSIBLY_INVALID	The dependent element may have changed.
MCS_NOTICES_ INVALID	MCS$K_NOTICES_ INVALID	The dependent element has changed.
MCS_NOTICES_ MSG_NEW_ VERSION	MCS$_NOTICES_ MSG_NEW_ VERSION	A new version of the dependent element has been created by either the NEW message or the REPLACE message. Metadata element types do not send notices for these reasons, however.

C Binding

```
MCS_STATUS MCS_read_notice (
               MCS_VALUE *notice_value,
               MCS_ELEMENTID *sender_elementid,
               MCS_LONGINT *cause_flag )
```

Description

The *MCS_read_notice* routine reads information from a single notice. Notices are sent to owners of dependency relationships when one of their member elements changes. The *MCS_check_notices* routine returns a list of the notices that have been received by an element.

Condition Values

ERRREADNOTICE	An error occurred while reading the notice structure.
MUSTBENOTICE	The value structure was not of data type MCS_NOTICE.
SUCCESS	Normal successful completion.

See Also

MCS_check_notices
MCS_clear_notices
MCS_force_notices

MCS_scan_dir

Creates and returns a scan of elements based on name (possibly including wildcards) and type.

OpenVMS Format

status = MCS$scan_dir scan_value, directory_name, elementtype_elementid

Arguments

scan_value
type: MCS$R_VALUE
access: write only
mechanism: by reference

Value structure filled in with the created scan. The application is responsible for allocating the value structure and must free the scan with *MCS_scan_free* after use.

directory_name
type: MCS$R_STRINGDSC
access: read only
mechanism: by descriptor

Name of element(s) to search for (may contain wildcards). Version numbers must be absolute, not relative, for this routine. Enclose the branch name and version number in parentheses.

elementtype_elementid
type: MCS$R_ELEMENTID
access: read only
mechanism: by reference

Element ID of the type to search for.

C Binding

```
MCS_STATUS MCS_scan_dir (
              MCS_VALUE *scan_value,
              MCS_STRING directory_name,
              MCS_ELEMENTID *elementtype_elementid )
```

Description

The *MCS_scan_dir* routine creates and returns a scan of elements whose names match the *directory_name* argument and whose type is the type specified by *elementtype_elementid* or is one of its subtypes. Because the *directory_name* argument may contain wildcards, the routine can find more than one element. After *MCS_scan_dir* has returned a scan, you can use the various scan manipulation routines to traverse the scan.

The *MCS_scan_dir* and *MCS_scan_query* routines both create scans that are not associated with the value of a property. Typically, you get a scan by sending GETPROP to an element for a scan-valued property. If you do not specify a version number, *MCS_scan_dir* uses the following default processing rules:

- returns the latest element in any collection under top
- if there is no version in any collection under top, returns the latest element in any partition

Condition Values

ERRSCANQUERY	An error occurred while trying to construct the scan; check the error stack for the cause.
MUSTABORT	Abort transaction; database corrupt.
ERR_NO_ACTIVE_TXN	There is no active transaction.
SUCCESS	Normal successful completion.

See Also

MCS_scan_query

MCS_scan_free

Frees a scan and associated memory. This routine is equivalent to *MCS_datatype_free*.

OpenVMS Format

```
status = MCS$scan_free   scan_value
```

Arguments

scan_value

type:	MCS$R_VALUE
access:	read/write
mechanism:	by reference

Value structure containing the scan to free.

C Binding

```
MCS_STATUS MCS_scan_free (
               MCS_VALUE *scan_value )
```

Description

See the description of *MCS_datatype_free*.

Condition Values

ERRSCANQUERY	An error occurred while trying to free the scan; check the error stack for the cause.
SUCCESS	Normal successful completion.

See Also

MCS_scan_new

MCS_scan_getByName

Finds the next element with a specified name in a scan and returns the element ID.

OpenVMS Format

```
status = MCS$scan_getByName  scan_value, element_name, element_elementid,
                             [relation_elementid]
```

Arguments

scan_value
type: MCS$R_VALUE
access: read/write
mechanism: by reference

Value structure containing the scan.

element_name
type: MCS$R_STRINGDSC
access: read only
mechanism: by descriptor

Name of the element to be returned. The name may include wildcard characters.

element_elementid
type: MCS$R_ELEMENTID
access: write only
mechanism: by reference

Element ID of the first element that matches the name.

relation_elementid
type: MCS$R_ELEMENTID
access: write only
mechanism: by reference

Optional. This argument is filled in with the ID of the relationship that connects *element_elementid* in this scan. The application is responsible for allocating the element ID.

MCS_scan_getByName

C Binding

```
MCS_STATUS MCS_scan_getByName (
               MCS_VALUE *scan_value,
               MCS_STRING element_name,
               MCS_ELEMENTID *element_elementid,
               MCS_ELEMENTID *relation_elementid )
```

Description

The *MCS_scan_getByName* routine finds the next element (with the specified name) in a scan and returns its element ID. The search starts with the element following the current position. If the scan has never been accessed or reset, *MCS_scan_getByName* starts the search with the first element in the scan. You can find only the first scan element by using this routine if the scan has never been accessed or if you call *MCS_scan_reset* to reset it.

If you specify the *relation_elementid* argument, the routine also returns the element ID of the relationship that connects the element in this scan (if this is a relation property). You can use this element ID if you need access to properties on the relationship, or if you want to pass it to *MCS_scan_remove* to remove the element from the scan.

If a wildcard is passed in for the name, this call returns the next element in the scan that matches the wildcard name.

Version numbers must be absolute, not relative, for this routine. Enclose the version number in parentheses.

If you do not specify a version number, *MCS_scan_getByName* uses the following default processing rules:

- returns the latest element in any collection under top
- if there is no version in any collection under top, returns the latest element in any partition

See the *Oracle CDD/Repository Architecture Manual* for more information on Oracle CDD/Repository names.

Condition Values

ENDFIND	No more elements in the scan matched the specified name.
ERRGETBYNAME	An error occurred while attempting to get the scan; check the error stack for the cause.
MUSTABORT	Abort transaction; database corrupt.
ERR_NO_ACTIVE_TXN	There is no active transaction.
SUCCESS	Normal successful completion.

MCS_scan_getCurrent

Returns the current element in a scan.

OpenVMS Format

status = MCS$scan_getCurrent scan_value, element_elementid, [relation_elementid]

Arguments

scan_value
type: MCS$R_VALUE
access: read/write
mechanism: by reference

Value structure containing the scan.

element_elementid
type: MCS$R_ELEMENTID
access: write only
mechanism: by reference

Returned element ID of the current element in the scan. The application is responsible for allocating the element ID.

relation_elementid
type: MCS$R_ELEMENTID
access: write only
mechanism: by reference

Optional. This argument is filled in with the ID of the relationship that connects *element_elementid* in this scan. The application is responsible for allocating the element ID.

C Binding

```
MCS_STATUS MCS_scan_getCurrent (
               MCS_VALUE *scan_value,
               MCS_ELEMENTID *element_elementid,
               MCS_ELEMENTID *relation_elementid )
```

Description

The *MCS_scan_getCurrent* routine returns the current element in the scan. The current element is the element most recently retrieved by one of the MCS_scan_get routines. If you call *MCS_scan_getCurrent* on a scan that has not previously been accessed, you receive an error.

If you specify the *relation_elementid* argument, the routine also returns the element ID of the relationship that connects the element in this scan. You can use this element ID if you need access to properties on the relationship, or if you want to pass it to *MCS_scan_remove* to remove the element from the scan.

If the scan is empty, the returned element ID and relation element ID are null.

Condition Values

ERRGETCURRENT	An error occurred while trying to get the current element; check the error stack for the cause.
MUSTABORT	Abort transaction; database corrupt.
ERR_NO_ACTIVE_TXN	There is no active transaction.
SUCCESS	Normal successful completion.

See Also

MCS_scan_getFirst
MCS_scan_getNext

MCS_scan_getFirst

Returns the first element in a scan and sets the current position at the first element.

OpenVMS Format

status = MCS$scan_getFirst scan_value, element_elementid, [relation_elementid]

Arguments

scan_value
type: MCS$R_VALUE
access: read/write
mechanism: by reference

Value structure containing the scan.

element_elementid
type: MCS$R_ELEMENTID
access: write only
mechanism: by reference

Element ID of the first element in the scan. The application is responsible for allocating the element ID.

relation_elementid
type: MCS$R_ELEMENTID
access: write only
mechanism: by reference

Optional. This argument is filled in with the ID of the relationship that connects *element_elementid* in this scan. The application is responsible for allocating the element ID.

C Binding

```
MCS_STATUS MCS_scan_getFirst (
               MCS_VALUE *scan_value,
               MCS_ELEMENTID *element_elementid,
               MCS_ELEMENTID *relation_elementid )
```

Description

The *MCS_scan_getFirst* routine resets the scan handle to the start of the scan and returns the first element in the scan, leaving the current position at the first element. If you specify the *relation_elementid* argument, the routine also returns the element ID of the relationship that connects the element in this scan. You can use this element ID if you need access to properties on the relationship, or if you want to pass it to *MCS_scan_remove* to remove the element from the scan.

Use this routine only to restart a scan traversal from the beginning. For a fresh scan (one that has not had any MCS_scan_get routine called on it or that has had *MCS_scan_reset* called on it), *MCS_scan_getNext* retrieves the first element, and is more efficient than *MCS_scan_getFirst*.

Condition Values

ENDFIND	The scan was empty.
ERRGETFIRST	An error occurred while trying to get the first element in the scan; check the error stack for the cause.
MUSTABORT	Abort transaction; database corrupt.
ERR_NO_ACTIVE_TXN	There is no active transaction.
SUCCESS	Normal successful completion.

See Also

MCS_scan_getCurrent
MCS_scan_getNext

MCS_scan_getNext

Returns the next element in a scan and advances the current position in the scan to this element.

OpenVMS Format

status = MCS$scan_getNext scan_value, element_elementid, [relation_elementid]

Arguments

scan_value
type: MCS$R_VALUE
access: read/write
mechanism: by reference

Value structure containing the scan.

element_elementid
type: MCS$R_ELEMENTID
access: write only
mechanism: by reference

Element ID of the next element in the scan. The application is responsible for allocating the element ID.

relation_elementid
type: MCS$R_ELEMENTID
access: write only
mechanism: by reference

Optional. This argument is filled in with the element ID of the relationship that connects *element_elementid* in this scan. The application is responsible for allocating the element ID.

C Binding

```
MCS_STATUS MCS_scan_getNext (
               MCS_VALUE *scan_value,
               MCS_ELEMENTID *element_elementid,
               MCS_ELEMENTID *relation_elementid )
```

Description

The *MCS_scan_getNext* routine returns the element following the current element in the scan, and advances the current element. If the scan has not been accessed before or has been reset, *MCS_scan_getNext* returns the first element in the scan. If you specify the *relation_elementid* argument, the routine also returns the element ID of the relationship that connects the element in this scan. You can use this element ID if you need access to properties on the relationship, or if you want to pass it to *MCS_scan_remove* to remove the element from the scan.

Condition Values

ENDFIND	The scan contains no more elements.
ERRGETNEXT	An error occurred while attempting to get the next element in the scan; check the error stack for the cause.
MUSTABORT	Abort transaction; database corrupt.
ERR_NO_ACTIVE_TXN	There is no active transaction.
SUCCESS	Normal successful completion.

See Also

MCS_scan_getCurrent
MCS_scan_getFirst

MCS_scan_insert

Adds an element to a scan.

OpenVMS Format

status = MCS$scan_insert scan_value, element_elementid

Arguments

scan_value
type: MCS$R_VALUE
access: read/write
mechanism: by reference

Value structure containing the scan.

element_elementid
type: MCS$R_ELEMENTID
access: read only
mechanism: by reference

Element ID of the element to insert.

C Binding

```
MCS_STATUS MCS_scan_insert (
               MCS_VALUE *scan_value,
               MCS_ELEMENTID *element_elementid )
```

Description

The *MCS_scan_insert* routine inserts an element into a scan that has been read from a property. Subsequently using SETPROP to assign the scan value back to the property creates a new relationship of the appropriate type to connect the inserted element with the element that possesses the scan property. The relationship is not created (and the new element does not appear in the scan) until SETPROP has been used. To see the effects of inserting an element, you must use SETPROP to set the property value followed by GETPROP to retrieve it.

Condition Values

SUCCESS	Normal successful completion.

See Also

MCS_scan_insert_with_args
MCS_scan_remove

MCS_scan_insert_with_args

Inserts an element into a scan and supplies initial values for properties on the resulting relationship.

OpenVMS Format

status = MCS$scan_insert_with_args scan_value, element_elementid, argument_list_value

Arguments

scan_value
type: MCS$R_VALUE
access: read/write
mechanism: by reference

Value structure containing the scan.

element_elementid
type: MCS$R_ELEMENTID
access: read only
mechanism: by reference

Element ID of the element to insert.

argument_list_value
type: MCS$R_VALUE
access: read only
mechanism: by reference

Value structure containing the argument list of properties to set on the new relationship. The application is responsible for freeing this list following the call to scan_insert_with_args. (See the description of SETPROP for a complete description of this argument's format.)

C Binding

```
MCS_STATUS MCS_scan_insert_with_args (
               MCS_VALUE *scan_value,
               MCS_ELEMENTID *element_elementid,
               MCS_VALUE *argument_list_value )
```

Description

The *MCS_scan_insert_with_args* routine inserts an element into a scan and additionally sets properties on the resulting relationship. (See *MCS_scan_insert* for a description of how this operation creates relationships.) Supply the property values in an argument list, just as you would when sending the NEW or SETPROP message.

You must use this routine when the relationship that results from the scan insertion has required properties.

MCS_scan_insert_with_args makes a copy of the argument list passed to it in the *argument_list_value* argument. Therefore, your application can free the list immediately after calling *MCS_scan_insert_with_args*.

Condition Values

ERRSCANINS	An error occurred while trying to insert the element or set property values on the relationship; check the error stack for the cause.
SUCCESS	Normal successful completion.

See Also

MCS_scan_insert
MCS_scan_remove

MCS_scan_new

Creates a new scan.

OpenVMS Format

status = MCS$scan_new scan_value, [element_elementid,] [scan_comp_rtn,] [property_name,] [property_owns_type,] [method_ctx]

Arguments

scan_value
type: MCS$R_VALUE
access: write only
mechanism: by reference

Value structure filled in with the new scan.

element_elementid
type: MCS$R_ELEMENTID
access: read only
mechanism: by reference

Optional. Element ID of the element that owns this property.

scan_comp_rtn
type: procedure entry mask
access: read only
mechanism: by reference

Optional. Entry point to a scan computation routine that you write and that Oracle CDD/Repository calls when it needs to access the scan. See the Description section for the calling sequence of this routine.

property_name
type: MCS$R_STRINGDSC
access: read only
mechanism: by descriptor

Optional. Name of the property for which this scan is (or will be) the value.

property_owns_type
type: MCS$R_ELEMENTID
access: read only
mechanism: by reference

Optional. Element ID of the element type that defines the property for which this scan is (or will be) the value.

method_ctx

type: unspecified
access: read only
mechanism: by reference

Optional. Pointer to an area of context information maintained for the use of a scan computation routine. This argument is read only with respect to the *MCS_scan_new* routine, but read/write with respect to the scan computation routine.

C Binding

```
MCS_STATUS MCS_scan_new (
               MCS_VALUE *scan_value,
               MCS_ELEMENTID *element_elementid,
               MCS_STATUS (*scan_comp_rtn)(),
               MCS_STRING property_name,
               MCS_ELEMENTID *property_owns_type,
               void *method_ctx )
```

Description

The *MCS_scan_new* routine creates and returns a scan.

Typically, an application gets a scan by sending the GETPROP message for a scan-valued property. However, there are two situations in which an application must use *MCS_scan_new*:

- The application needs to create a scan to supply with the NEW message as the initial value for a property.
- The application defines a computed property whose value is a scan; the method that is called to compute the property value must call *MCS_scan_new* to create the scan.

Which arguments you supply with *MCS_scan_new* depends on your use of the call, as follows:

- If you are creating a scan to supply as the initial value for a property, supply values for the following arguments only:

  scan_value
  property_name
  property_owns_type

- If you are creating a scan for a computed scan property, supply values for the following arguments only:

  scan_value
  element_elementid
  scan_comp_rtn
  property_name
  method_ctx

Condition Values

ERRSCANNEW	An error occurred while trying to create the new scan; check the error stack for the cause.
MUSTABORT	Abort transaction; database corrupt.
ERR_NO_ACTIVE_TXN	There is no active transaction.
SUCCESS	Normal successful completion.

See Also

MCS_scan_free

MCS_scan_query

Dynamically creates scans.

OpenVMS Format

status = MCS$scan_query scan_value, source_elementid, query_buff

Arguments

scan_value
type: MCS$R_VALUE
access: read/write
mechanism: by reference

Value structure for the returned scan.

source_elementid
type: MCS$r_ELEMENTID
access: read only
mechanism: by reference

Starting point for the query.

query_buff
type: MCS$R_STRINGDSC
access: read only
mechanism: by descriptor

Dictionary query buffer. (See Section C.5 for more information.)

C Binding

```
MCS_STATUS MCS_scan_query (
                MCS_VALUE *scan_value,
        MCS_ELEMENTID *source_elementid,
                MCS_STRING query_buffer )
```

MCS_scan_query

Description

The *MCS_scan_query* routine creates dynamic scans. It allows users of the repository to find information based on user-supplied information including relationship name, from-element, and to-element. (See Appendix C for information on using routines with buffers.)

Condition Values

MUSTABORT	Abort transaction; database corrupt.
ERR_NO_ACTIVE_TXN	There is no active transaction.
SUCCESS	Normal successful completion.

MCS_scan_remove

Removes an element from a scan.

OpenVMS Format

status = MCS$scan_remove scan_value, [element_elementid,] [relation_elementid]

Arguments

scan_value
type: MCS$R_VALUE
access: read/write
mechanism: by reference

Value structure containing the scan.

element_elementid
type: MCS$R_ELEMENTID
access: read only
mechanism: by reference

Optional. Element ID of the element to remove. You must specify either this argument or the *relation_elementid* argument, but not both.

relation_elementid
type: MCS$R_ELEMENTID
access: read only
mechanism: by reference

Optional. Element ID of the relationship to remove. You must specify either this argument or the *element_elementid* argument, but not both.

C Binding

```
MCS_STATUS MCS_scan_remove (
               MCS_VALUE *scan_value,
               MCS_ELEMENTID *element_elementid,
               MCS_ELEMENTID *relation_elementid )
```

MCS_scan_remove

Description

The *MCS_scan_remove* routine removes an element from a scan that has been read from a property. Subsequently using SETPROP deletes the relationship that connects the removed element with the element that possesses the scan property. The relationship is not deleted (and the removed element does not disappear from the scan) until SETPROP has been used. To see the effects of removing an element, you must use SETPROP to set the property value followed by GETPROP to retrieve it.

If you specify the *element_elementid* argument, the routine finds and deletes the associated relationship. If you specify the *relation_elementid* argument, the routine deletes the relationship directly. The effect is the same in either case, but the routine is more efficient if you specify the relationship instead of the element.

Condition Values

ERRSCANREMOVE	An error occurred while attempting to remove the element from the scan; check the error stack for the cause.
MUSTABORT	Abort transaction; database corrupt.
ERR_NO_ACTIVE_TXN	There is no active transaction.
SUCCESS	Normal successful completion.

See Also

MCS_scan_insert
MCS_scan_insert_with_args

MCS_scan_reset

Resets a scan to the beginning.

OpenVMS Format

```
status = MCS$scan_reset   scan_value
```

Arguments

scan_value

type:	MCS$R_VALUE
access:	read/write
mechanism:	by reference

Value structure containing the scan.

C Binding

```
MCS_STATUS MCS_scan_reset (
                MCS_VALUE *scan_value )
```

Description

The *MCS_scan_reset* routine resets a scan to the beginning. Following this call, call *MCS_scan_getNext* to return the first element in the scan, then call *MCS_scan_getByName* to search the scan starting with the first element. (By comparison, call *MCS_scan_getFirst* to return the first element in the scan, then call *MCS_scan_getNext* or *MCS_scan_getByName* to start accessing the scan at its second element.)

Condition Values

SUCCESS	Normal successful completion.

MCS_session_initiate

Establishes a repository session.

OpenVMS Format

```
status = MCS$session_initiate   session_handle
```

Arguments

session_handle

type: MCS$R_SESSION
access: write only
mechanism: by reference

Returned session handle. The application is responsible for allocating the session handle.

C Binding

```
MCS_STATUS MCS_session_initiate (
               MCS_SESSION *session_handle )
```

Description

The *MCS_session_initiate* routine establishes a repository session and returns a session handle, which is a unique identifier for the session. All subsequent calls to begin and end a transaction, open a repository, or terminate the session must refer to the session handle. An application can have only one session active at a time, although multiple sessions can access the same repository.

Use *MCS_session_initiate* with the following routines:

- *MCS_session_transaction_init*—To start a transaction
- *MCS_session_transaction_term*—To end a transaction
- *MCS_initiate_database*—To access a repository
- *MCS_session_terminate*—To end a session

Condition Values

ERRSTARTSES	An error occurred while attempting to start the session; check the error stack for the cause.
SUCCESS	Normal successful completion.

See Also

MCS_session_terminate

MCS_session_terminate

Frees all memory associated with a session.

OpenVMS Format

```
status = MCS$session_terminate   session_handle
```

Arguments

session_handle

type: MCS$R_SESSION
access: read/write
mechanism: by reference

Session handle that was returned from a session_initiate call.

C Binding

```
MCS_STATUS MCS_session_terminate (
               MCS_SESSION *session_handle )
```

Description

The *MCS_session_terminate* routine frees all memory associated with a session. Use it when referring to a specific session instead of the global session.

When a session terminates, all element IDs used during the session become invalid; they cannot be reused in another session. However, values that were allocated during the session remain valid.

Use *MCS_session_terminate* with the following routines:

- *MCS_session_initiate*—To start a session
- *MCS_session_transaction_init*—To start a transaction
- *MCS_session_transaction_term*—To end a transaction
- *MCS_initiate_database*—To access a repository

Condition Values

ERRTERMINATE	An error occurred while trying to terminate the session; check the error stack for the cause.
SUCCESS	Normal successful completion.

See Also

MCS_session_initiate

MCS_session_transaction_init

Begins a transaction within a specific session.

OpenVMS Format

status =MCS$session_transaction_init existing_transaction_handle, new_transaction_handle, session_handle, [read_only_flag]

Arguments

existing_transaction_handle
type: MCS$R_TRANSACTION
access: read only
mechanism: by reference

Transaction handle. If the handle is 0, the routine starts a new transaction; if the handle is an existing transaction handle, the routine starts a subtransaction.

Note

Subtransactions are not currently supported. This argument must be 0.

new_transaction_handle
type: MCS$R_TRANSACTION
access: write only
mechanism: by reference

Returned transaction handle for the new transaction. Use this handle in the call to *session_transaction_term* that ends this transaction.

session_handle
type: MCS$R_SESSION
access: read only
mechanism: by reference

Handle of session within which to start the transaction.

read_only_flag

type: MCS$L_BOOLEAN
access: read only
mechanism: by reference

Optional. Specifies whether the transaction is a read-only transaction or a read/write transaction. Read/write is the default. The values for the *read_only_flag* argument are as follows (C binding/OpenVMS binding):

C Binding	OpenVMS Binding	Meaning
MCS_TRANSACTION_READWRITE	MCS$K_TRANSACTION_READWRITE	Read/write transaction
MCS_TRANSACTION_READONLY	MCS$K_TRANSACTION_READONLY	Read-only transaction

C Binding

```
MCS_STATUS MCS_session_transaction_init (
               MCS_TRANSACTION *existing_transaction_handle,
               MCS_TRANSACTION *new_transaction_handle,
               MCS_SESSION *session_handle,
               MCS_BOOLEAN read_only_flag )
```

Description

The *MCS_session_transaction_init* routine begins a transaction within a specific session. A transaction groups operations that must all succeed or fail as a unit. When you end a transaction (using *MCS_session_transaction_term*), you can choose to abort the transaction and roll back the changes that were made in the repository. If a system failure occurs during a transaction, the changes are rolled back automatically when the system restarts.

Any repository activity that manipulates elements requires that a transaction be active. This includes opening a repository by using *MCS_initiate_database.*

This version of Oracle CDD/Repository does not support subtransactions. If *existing_transaction_handle* has a nonzero value, an error is returned.

When you start a transaction that reads but does not modify the repository, specify the *read_only_flag* argument as TRANSACTION_READONLY. Oracle CDD/Repository uses this information to reduce locking contention in the repository.

MCS_session_transaction_init

Use *MCS_session_transaction_init* with the following routines:

- *MCS_session_initate*—To start a session
- *MCS_session_transaction_term*—To end a transaction
- *MCS_initiate_database*—To access a repository
- *MCS_session_terminate*—To end a session

Condition Values

ERRSTARTTXN	An error occurred while attempting to start the transaction; check the error stack for the cause.
SUCCESS	Normal successful completion.

See Also

MCS_session_transaction_term

MCS_session_transaction_term

Ends a transaction within a session by either making the changes permanent or rolling them back.

OpenVMS Format

status = MCS$session_transaction_term transaction_handle, [abort_flag]

Arguments

transaction_handle

type: MCS$R_TRANSACTION
access: read only
mechanism: by descriptor

Transaction handle of the transaction to end.

abort_flag

type: MCS$L_BOOLEAN
access: read only
mechanism: by reference

Optional. If specified with a value of MCS_TRANSACTION_ABORT (MCS$K_TRANSACTION_ABORT), the changes made during the transaction are rolled back. If omitted or specified with a value of MCS_TRANSACTION_COMMIT (MCS$K_TRANSACTION_COMMIT), the changes are made permanent in the repository.

C Binding

```
MCS_STATUS MCS_session_transaction_term (
               MCS_TRANSACTION *transaction_handle,
               MCS_BOOLEAN abort_flag )
```

Description

The *MCS_session_transaction_term* routine ends a transaction within a session by either making the changes permanent or rolling them back.

Use *MCS_session_transaction_term* with the following routines:

- *MCS_session_initiate*—To start a session

- *MCS_session_transaction_init*—To start a transaction
- *MCS_initiate_database*—To access a repository
- *MCS_session_terminate*—To end a session

Condition Values

ERRTERMINATETXN	An error occurred while attempting to end the transaction; check the error stack for the cause.
MUSTABORT	Abort transaction; database corrupt. (Can be returned if abort_flag is not specified or is FALSE.)
ERR_NO_ACTIVE_TXN	There is no active transaction.
SUCCESS	Normal successful completion.

See Also

MCS_session_transaction_init

MCS_set_default

Sets the default directory within the repository.

OpenVMS Format

status = MCS$set_default directory_name, session_handle

Arguments

directory_name
type: MCS$R_STRINGDSC
access: read only
mechanism: by descriptor

Path name.

session_handle
type: MCS$R_SESSION
access: read only
mechanism: by reference

Session handle of session within which to set default.

C Binding

```
MCS_STATUS MCS_set_default (
               MCS_STRING directory_name,
               MCS_SESSION *session_handle)
```

Description

The *MCS_set_default* routine sets the repository default to the specified directory path. (You also can use the logical CDD$DEFAULT to set the repository default to the specified directory path.)

Condition Values

ERRDIR_SET_DEFAULT	An error occurred while attempting to set default; check the error stack for the cause.
SUCCESS	Normal successful completion.

A
Error Handling

This appendix describes the following topics:

- format of the error stack
- manipulation of the error stack
- errors that occur during message processing

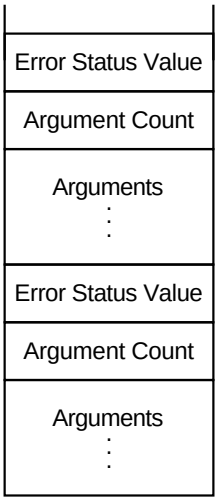
Oracle CDD/Repository methods report errors on an error stack. There is one error stack for each repository process.

The error stack is accessed through a set of error stack manipulation routines. These routines interpret and use the error-handling functions provided by the system on which the repository is running. Your methods must use the same mechanisms if they are to coordinate with the methods supplied with Oracle CDD/Repository.

A.1 Format of the Error Stack

The **error stack** is a list of error entries that methods can read and manipulate. Each entry on the error stack consists of a status value, an argument count, and a list of arguments used for formatting an error message, as shown in Figure A–1.

Figure A–1 The Error Stack



ZK–2957A–GE

A.2 Manipulating the Error Stack

The following routines manipulate the error stack. These calls are most useful if you write methods or refine the methods supplied with Oracle CDD/Repository. The calls also are useful if you write applications that reference messages.

- *MCS_errorstack_set* pushes an entry onto the error stack. You must supply the error status code, the number of arguments, and a value structure containing the list of arguments.
- *MCS_errorstack_getCurrentSize* returns the current size of the error stack.
- *MCS_errorstack_getStatus* returns the status code for an error stack entry. You use an index number, with 0 equal to the top entry, to specify the entry.
- *MCS_errorstack_format* creates a string containing the formatted error message for a specified error stack entry. You specify the entry by its index.
- *MCS_errorstack_clear* removes a specified entry from the error stack. *MCS_errorstack_clearAll* removes all entries.
- *MCS_errorstack_getMaxSize* and *MCS_errorstack_setMaxSize* allow you to determine the maximum size of the error stack.

This list shows that the error stack does not have a true stack structure. Although you can place entries only on top, you can read and clear entries from anywhere in the structure. (For the syntax of the Oracle CDD/Repository error stack routines, see Chapter 8.)

A.3 Errors During Message Processing

In the object-oriented programming environment, the *MCS_dispatch_op* routine invokes the appropriate method for each combination of element and message. That method may in turn call the method of its supertype or dispatch an entirely different message to complete its processing. Errors can occur at any point during the processing of a method, its supertype methods, or the other methods that it invokes.

Errors also can occur at any level in the processing. For example, an error may be reported by the operating system or the database on which Oracle CDD/Repository is layered.

The error stack allows error reporting from all invoked methods and from all levels of the implementation. Code that detects an error pushes an entry onto the error stack and returns an error status. Code that receives an error status from a routine can examine the entries on the stack to determine the cause of the error. Before returning, it can choose to handle the error and clear the entry, defer error handling to the calling routine, or add its own entry to the stack. At the user interface level, an application can display the contents of the error stack or it can clear the stack.

The dispatcher does not put errors returned by methods on the error stack. It assumes the method has done that. However, it does check the status returned by the method and terminates the dispatch request if that status is not SUCCESS. If a method returns a bad status, the dispatcher does not try to execute any postambles associated with the method.

If the method called by *MCS_dispatch_op* contains a call to *MCS_dispatch_superOp* (as shown in the following example), the dispatcher considers the call to be a separate request:

```
method:
     status = mcs$dispatch_superOp
```

If the call is not successful, the dispatcher terminates this request and returns the error status to your method. You must then return this status in the method routine if you want the status to be returned by the original *MCS_dispatch_op* call.

A.3.1 Handling Errors

Methods and calls to MCS routines can be successful or unsuccessful. A status value can be one of the following types:

- Success and informational—The routine completed successfully.
- Warning—The routine completed successfully, but you should be aware of a potential problem.
- Error or Fatal—The routine could not complete.

Your methods should not place success, informational, and warning status values on the error stack. You should place only error and fatal status values on the stack.

A.3.2 Handling Success, Informational, and Warning Status Values

A return status of SUCCESS clearly indicates a successful outcome, and the error stack typically is left empty. If a method is invoked from inside another method, there may be information on the error stack relevant to the other method. To be sure your method does not inadvertently clear the error stack of relevant information, a method should add only to the error stack and the client application should clear the stack, when appropriate. Use *MCS_errorstack_clear* to clear individual entries from the stack. Use *MCS_errorstack_clearAll* to clear all entries from the stack.

For some MCS routines, other status codes also may indicate success. For example, a return code of ARGNOTFOUND from *MCS_arglist_findArg* is not an error if you are looking for an optional argument. ENDFIND from *MCS_scan_getNext* is not an error unless you are expecting another member in the scan. Informational or warning status values such as these are not placed on the stack. Methods that you write should not place them on the stack, either. If your routine detects one of these values, it must decide if the status indicates an error and process it accordingly.

A.3.3 Handling Error and Fatal Status Values

Error and fatal status codes indicate an error that prevents successful completion of the method. These types of errors include the absence of a required argument on the message argument list, access errors, and so on. In general, only errors that prevent successful completion of the method are placed on the error stack.

If one of your methods fails, it should place the status value and the list of arguments on the error stack, using *MCS_errorstack_set*. The method should then return the error status to the calling routine.

For example:

```
if( /* something happens here that is an error */ ) {
    mcs$errorstack_set( status, 0, arglist );
    return( status );
}
```

Example A–1 shows how a method or application performs error handling using the error stack routines. It formats and clears each entry in the error stack.

Example A–1 Displaying the Contents of the Error Stack

```
MCS_STRING buffer;
MCS_LONGINT index, size;

for( index = 0, mcs$get_current_size( size );
     size > 0;
     index++, --size ) {
       mcs$errorstack_format( index, buffer );
       mcs$errorstack_clear( index );
       fprintf( stderr, "%s\n", buffer );
}
```

A.3.4 Reading from the Error Stack

If your method receives a return value other than SUCCESS, it should examine the contents of the error stack to determine what caused the error. *MCS_errorstack_getStatus* lets you retrieve a status value from the stack.

The method can attempt to recover the error or display some or all of the error stack. Each status value returned by a method has a message associated with it. *MCS_errorstack_format* formats the message for an error stack entry.

A.3.5 Putting Messages on the Error Stack

If your function detects an error, it should push one or more error status codes on the error stack, and should return the last error status that it pushed on the error stack.

Example A–2 illustrates how a method function detects an illegal value and pushes an application-specific status code onto the error stack, followed by the general status code SETPROPFAILED. It then returns SETPROPFAILED.

Example A–2 Returning an Error Status

```
if ( ( yPropValue >= YPROP_VALUE1 ) &&
     ( yPropValue <= YPROP_VALUE_LAST ) ) {
    break;          /* Value OK, dispatch_superOp */
    }
else {              /* Value out-of-bounds */
    /* Set the status on the yProp argspec */
    MCS_arglist_setIndexValue (
        &propertyList,
        yPropIndex,
        0,          /* Do not change the property value */
        YPROP_BADVALUE);
    /* Push error codes on error stack */
    MCS_errorstack_set (
        YPROP_BADVALUE, 0, 0);
    MCS_errorstack_set (
        MCS_SETPROPFAILED, 0, 0);
    /* Finally, return SETPROPFAILED */
    return ( MCS_SETPROPFAILED );
    }
```

Example A–3 Returning a Memory Allocation Error

```
stateInfo = (CHILDREN_WITH_A_STATE_INFO *)malloc(
    sizeof(CHILDREN_WITH_A_STATE_INFO) );

if ( !stateInfo ) {
    status = MCS_BADMALLOC;
    MCS_errorstack_set(status,0,0);
    MCS_arglist_setNameValue(
        &propertyList,
        "childrenWithA",
        0,
        status);
    return ( status );
    }
```

In Example A–3, the method function attempts to allocate memory and fails. Because this is an error detected by the method function rather than by a routine that it calls, the method function is responsible for placing the error status on the error stack and then returning that status.

Note

Example A–2 and Example A–3 return error status codes defined by Oracle CDD/Repository. Your method should attempt to return a standard status if possible.

A.3.6 Unexpected Errors

If your function receives an unexpected status code from an Oracle CDD/Repository callable interface routine, the function should return that status code without placing anything on the error stack. The routine that detected the error has already placed information on the error stack, so your method does not have to.

A method function should always check the return status of any Oracle CDD/Repository callable interface routine it calls, unless it is already in the process of reporting an error. For example:

```
stateInfo = (CHILDREN_WITH_A_STATE_INFO *)malloc(
    sizeof(CHILDREN_WITH_A_STATE_INFO) );

if ( !stateInfo ) {
    status = MCS_BADMALLOC;
    MCS_errorstack_set(status,0,0);
    MCS_arglist_setNameValue(
        &propertyList,
        "childrenWithA",
        0,
        status);
    return ( status );
    }
```

The call to malloc in this example attempts to do useful work. Its return status is tested and, if it is zero (indicating failure), calls to *MCS_errorstack_set* and *MCS_arglist_setNameValue* report the error. The status returns from these calls are not checked because they are part of the error reporting code. The return statement returns the status code from the malloc call as the value of the method function.

A.3.7 Setting the Status Field in an Argument List

The dispatch lists of the SETPROP, NEW, and GETPROP messages all use an *arglist* argument to specify a list of properties to be set or read. Each argument specification contains the name of a property, a value, and a status field. The status field records the success or failure of the methods for that property.

After one of these methods completes successfully, Oracle CDD/Repository places a success status in the status fields of each argument specification in the property list. But if the method does not complete successfully, one or more of the status fields contain a non-SUCCESS status indicating what went wrong with the operation for that property.

If your method uses SETPROP or NEW, it does not need to place the SUCCESS code in the status field to indicate a successful operation. The method that sets the property value performs that function. Your method does need to check for the SETPROPFAILED status on the error stack (the status may not be on the top of the stack). Your method also should check for a specific problem code in the status field of the argument specification that contains the offending property. If errors are found, the method must do the following:

1. Select a status code that indicates the nature of the error. The status code may be an application-specific code, or an unexpected return status from an Oracle CDD/Repository callable interface routine.
2. Determine the property that was being processed when the error occurred.
3. Place the status code in the status field of the argument specification that contains the property.
4. Call *MCS_errorstack_set* to add this same status code to the error stack, followed by SETPROPFAILED.
5. Return SETPROPFAILED.
6. It must *not* invoke *MCS_dispatch_superOp*.

For GETPROP, failure to return one or more property values results in the SOMEFAILED status on the error stack and specific problem codes in the status fields of the offending properties. Oracle CDD/Repository attempts to retrieve all the properties in the property list despite the failure. If Oracle CDD/Repository retrieves all the properties, the method places the SUCCESS code in the status field at the same time it places the property value in the value field. It then returns SUCCESS. If Oracle CDD/Repository cannot retrieve all the properties, the method must place the failure code in the status field and return the same failure code. The method should not return SOMEFAILED.

B

Utility Routines

This appendix describes the support utility routines you use to perform the following operations:

- translate directory names
- verify the integrity of a repository
- return the version of a repository
- check for messages
- call CDO from a program

Each section describes one call in detail and includes:

- the name of the entry point
- the parameters you exchange with Oracle CDD/Repository during the call, including each parameter's data type, read/write status, and passing mechanism
- some of the more frequent return values for the call
- suggestions for using the call

Calls are listed in alphabetical order.

CDD$TRANSLATE

Applies Oracle CDD/Repository name translation rules to a set of user-supplied directory names, returning fully translated names. No action is taken with the names. (See the *Oracle CDD/Repository Architecture Manual* for more information on Oracle CDD/Repository names.)

Format

CDD$TRANSLATE exception_vector, user_handle, dir_info_in, dir_info_out

Returns

OpenVMS usage:	cond_value
type:	longword (unsigned)
access:	write only
mechanism:	by value

Longword condition value. All Oracle CDD/Repository support utility routines return (by immediate value) a condition value in R0. Condition values that can be returned by this service include:

SS$_NORMAL	Operation completed successfully.

Arguments

exception_vector

OpenVMS usage:	message vector
type:	array of 20 longwords
access:	write only
mechanism:	by reference

An array of 20 longwords conforming to the OpenVMS exception vector format.

user_handle

OpenVMS usage:	quadword_unsigned
type:	quadword (unsigned)

access:	read only
mechanism:	by reference

An unsigned quadword identifying the user context within which Oracle CDD/Repository creates the session from CDD$ATTACH.

dir_info_in

OpenVMS usage:	unstructured
type:	text
access:	read only
mechanism:	by descriptor (static or dynamic)

A buffer identifying the directory entries whose names are to be translated.

dir_info_out

OpenVMS usage:	unstructured
type:	text
access:	read only
mechanism:	by descriptor (static or dynamic)

A buffer into which CDD$TRANSLATE places the translated names.

Description

If the directory name list passed as part of the input buffer is empty, the output buffer returns all the translations of the current default.

Each directory name in the output directory name list is a fully translated name.

If CDD$K_TYPE is included in the input buffer, it is included in the output buffer.

CDD$TRANSLATE does not check whether the elements identified by the translated name really exist.

If the input buffer includes a search list, all names are returned in the output directory list.

This utility uses the directory information buffer (see Appendix C for information on this buffer).

CDD$VERIFY

Confirms and optionally restores the integrity of a repository or directory.

Format

CDD$VERIFY exception_vector, database_dsc, actions, [session_handle], [msg_routine], [actprm]

Returns

OpenVMS usage:	cond_value
type:	longword (unsigned)
access:	write only
mechanism:	by value

Longword condition value. All Oracle CDD/Repository support utility routines return (by immediate value) a condition value in R0. Condition values that can be returned by this service include:

SS$_NORMAL	Operation completed successfully.
CDD$_DICINVALID	There were one or more problems with the repository that were not fixed.
CDD$_FIXED	There were one or more problems with the repository that were fixed.

Arguments

exception_vector

OpenVMS usage:	message vector
type:	array of 20 longwords
access:	write only
mechanism:	by reference

An array of 20 longwords conforming to the OpenVMS exception vector format.

database_dsc

OpenVMS usage:	text
type:	text string

access:	read only
mechanism:	by descriptor

A descriptor naming the repository to check for correctness.

actions

OpenVMS usage:	bit vector
type:	longword
access:	read only
mechanism:	by reference

A bit vector indicating what actions to take. See Table B–1 for a list of actions.

session_handle

OpenVMS usage:	quadword_unsigned
type:	quadword (unsigned)
access:	read only
mechanism:	by reference

Session handle to use while verifying the repository.

msg_routine

OpenVMS usage:	procedure
type:	procedure entry mask
access:	function call (before return)
mechanism:	by reference, procedure reference

A routine to process messages.

actprm

OpenVMS usage:	user_arg
type:	unspecified
access:	read only
mechanism:	by reference

The action parameter passed to CDD$VERIFY. It is passed to the msg_routine without any kind of verification.

Description

Exception_vector and actprm are parameters for msg_routine. Oracle CDD/Repository ignores any status value returned by msg_routine.

To verify your repository, call CDD$VERIFY with a session handle, no message routine, and with the bit cdd$m_vf_location set. CDD$VERIFY returns SS$_ NORMAL if the repository is in the right place, CDD$_DICINVALID otherwise.

If you plan to call CDD$VERIFY with the cdd$m_vf_fix bit set in the actions parameter, make sure your session is a long transaction.

Table B–1 describes the values that may be specified with the actions parameter.

Table B–1 Values Used with the Actions Parameter

Value	Meaning
cdd$m_vf_fix	If set, fix any problems as they are found.
cdd$m_vf_log	If set, report any successful checks as well as unsuccessful ones.
cdd$m_vf_location	Check that the repository's internal idea of where it is matches where it actually is. If cdd$m_vf_fix is set, the repository is updated to recognize its new location.
cdd$m_vf_ext_ref	Check that other databases point to the correct location for this database. If cdd$m_vf_fix and cdd$m_vf_location are set, this should be set also; the other databases will be modified to point to the new location.
cdd$m_vf_out_ref	Check that databases referred to by this database are in their correct locations. cdd$m_vf_fix has no effect on this flag.
cdd$m_vf_xdb_rel	Check that relations that cross databases are consistent and that owners of dependency relationships that cross databases have local copies of all the right members. If cdd$m_vf_fix and cdd$m_vf_xdb_rel are set, cdd$m_vf_out_ref should be set also. If cdd$m_vf_ fix is set, the most recent copy of the relationship is set to match the less recent copy.

(continued on next page)

Table B–1 (Cont.) Values Used with the Actions Parameter

Value	Meaning
cdd$m_vf_orphans	Check for orphans in the given database. If orphans are found and cdd$m_vf_fix is set, enter names for the orphans under anchor-name:CDD$ORPHANS. The names for the orphans are the processing name, if any, modified to be a reasonable directory name, or ORPHAN_n, if there is no processing name.
cdd$m_vf_names	Check names in the given database to ensure that they correspond to real objects. If cdd$m_vf_fix is specified, names that do not correspond to objects are removed.
cdd$m_vf_recovery	Check whether the database needs recovering and indicate which other databases it needs to recover. Specifying this option alone prevents the repository from trying to recover this database. Specifying any other option will cause recovery to be attempted; if recovery is successful, this option will produce a message indicating recovery is not necessary.

CDD$VERSION

Returns a formatted buffer of the versions of all protocols in the attached repository.

Format

CDD$VERSION exception_vector, session_handle, version_buffer

Returns

OpenVMS usage:	cond_value
type:	longword (unsigned)
access:	write only
mechanism:	by value

Longword condition value. All Oracle CDD/Repository support utility routines return (by immediate value) a condition value in R0. Condition values that can be returned by this service include:

SS$_NORMAL	Operation completed successfully.
CDD$_error	The versions could not be returned.

Arguments

exception_vector

OpenVMS usage:	message_vector
type:	array of 20 longwords
access:	write only
mechanism:	by reference

An array of 20 longwords conforming to the OpenVMS exception vector format.

session_handle

OpenVMS usage:	quadword_unsigned
type:	quadword (unsigned)
access:	read only
mechanism:	by reference

An unsigned quadword identifying the repository session in which Oracle CDD/Repository checks for versions.

version_buffer

OpenVMS usage:	unstructured
type:	text
access:	write only
mechanism:	by descriptor (static or dynamic)

A buffer to receive the versions of the protocols being used in the repositories invoked for this session.

The version buffer has the following format:

```
<version_buffer>    ::=     <block_header>
                            <versions>
                            <block_terminator>

<block_header>      ::=    CDD$K_VERSION_BUF_DSC
                           <major_version_number>
                           <minor_version_number>

<versions>          ::= CDD$K_LITERAL
                        DSC$K_DTYPE_T <word_length> <byte_string>
                        <version_list>

<version_list>   ::=  CDD$K_ODS_VERSION_LIST
                      [ CDD$K_ODS_NAME
                        <word_length>
                        <byte_string>
                        CDD$K_ODS_VERSION
                        <ods_major_version> <ods_minor_version> ]...

<major_version_number> ::= unsigned longword
<minor_version_number> ::= unsigned longword
<byte_string>          ::= repository name
<ods_major_version>    ::= unsigned longword
<ods_minor_version>    ::= unsigned longword
<word_length>          ::= unsigned word
<byte_string>          ::= Zero or more contiguous bytes.
```

The fields of the version buffer have the following meaning:

- major_version_number, minor_version_number

  A pair of unsigned longwords giving the version number of the version buffer. The present value is 1.0.

- version_list

  A list of software versions and names of the attached repositories.

- ods_major_version, ods_minor_version

  A pair of unsigned longwords identifying the protocol version of each attached repository.

Description

If there is no repository attached, the version buffer is empty.

CDO$CHECK_MESSAGES

Checks to see if there are messages on any of the named entities.

Format

CDO$CHECK_MESSAGES entity [,...]

Returns

OpenVMS usage:	cond_value
type:	longword (unsigned)
access:	write only
mechanism:	by value

Longword condition value. All CDO routines return (by immediate value) a condition value in R0. Condition values that can be returned by this service include:

CDD$_MESS	Messages were found on the entity or entities.
CDD$_NOMESS	No messages were found on the entity or entities.

Arguments

entity

OpenVMS usage:	text string
type:	text
access:	read only
mechanism:	by descriptor

The address of a string descriptor pointing to the string containing the entity name.

Description

When you supply more than one entity name as an argument to this call, Oracle CDD/Repository checks each entity in turn until it finds messages. When a message has been found on an entity, Oracle CDD/Repository returns CDD$_MESS and stops checking the remaining entities for messages. If no messages are found on any of the entities, CDD$_NOMESS is returned.

CDO$INTERPRET

Allows you to call CDO from a program.

Format

CDO$INTERPRET context_var, command_dsc, [output_routine], [output_routine_param]

Returns

OpenVMS usage:	cond_value
type:	longword (unsigned)
access:	write only
mechanism:	by value

Longword condition value. All CDO routines return (by immediate value) a condition value in R0. Condition values that can be returned by this service include:

SS$_NORMAL	Operation completed successfully.

Arguments

context_var

OpenVMS usage:	unstructured
type:	longword logical (unsigned)
access:	read/write
mechanism:	by reference

A data structure that establishes the context for your CDO session.

command_dsc

OpenVMS usage:	unstructured
type:	text string
access:	read only
mechanism:	by descriptor

The descriptor containing the CDO command to be executed. The length of the descriptor cannot exceed 64 kilobytes.

output_routine

OpenVMS usage:	unstructured
type:	virtual address
access:	read only
mechanism:	by reference

The name of the routine that you want the output from the CDO$INTERPRET routine sent to. If you do not specify an output routine, output is sent to SYS$OUTPUT.

output_routine_param

OpenVMS usage:	unstructured
type:	virtual address
access:	read only
mechanism:	by reference

The parameter sent by the CDO$INTERPRET routine to the output routine.

Description

The first time you call CDO$INTERPRET, pass a context variable with a value of 0. Do not reinitialize the context variable after you have called CDO$INTERPRET the first time.

C
Buffers

A buffer is a formatted block of data that can be read and interpreted by both the calling program and the called routine. Buffers are not text, but a set of bytes with a data type of MCS_MEMBLOCK. Several properties have a buffer format (see the *Oracle CDD/Repository Information Model Volume I* manual for information on a specific property). You also can use buffers with the *MCS_scan_query* routine to query a repository (see Chapter 8 for more information).

The buffers passed to the callable routines are dynamic or static strings passed by descriptor. Static string descriptors may generate overflow errors on output if the requested information does not fit in the string.

Oracle CDD/Repository buffers are formatted byte strings containing the information the repository system needs to process your request. The format of the buffer varies according to the type of buffer and whether the information is being passed to Oracle CDD/Repository for processing or returned to your program by Oracle CDD/Repository.

For example, when you retrieve an element from Oracle CDD/Repository, you provide a buffer describing the information you want. Oracle CDD/Repository reads the internal data structures describing the element, translates them into the format prescribed by the buffer, and inserts them into the buffer. If some information is unavailable, Oracle CDD/Repository inserts error information into the buffer along with the information that was available.

Buffers can contain other buffers.

C.1 Buffer Format

The following production rules, expressed in Backus-Naur Form, describe the complete format of all buffers.

```
<buffer>               ::= <block_header>
                           <body>
                           <block_terminator>

<block_header>         ::= <beginning>
                           <type>
                           <version>

<beginning>            ::= CDD$K_BEGIN

<type>                 ::= unsigned longword

<version>              ::= <major_version_number>
                           <minor_version_number>

<major_version_number> ::= unsigned longword
<minor_version_number> ::= unsigned longword
<body>                 ::= sequence of bytes
<block_terminator>     ::= CDD$K_EOC
```

CDD$K_BEGIN and CDD$K_EOC are both unsigned byte values defined in the Oracle CDD/Repository library files.

The following major sections of a buffer format are of special importance:

- Block header—Indicates the start of the buffer and the name and version number of the buffer.
- Body—Contains formatted information for the repository.
- Block terminator—Marks the end of the buffer.

C.1.1 Block Header

The block header identifies the kind of information that the buffer contains. It consists of the unsigned byte CDD$K_BEGIN, followed by a longword specifying the buffer type and two longwords identifying the buffer's version.

By examining the buffer type and version, your program can determine the structure of the information in the body of the buffer.

Subsequent versions of Oracle CDD/Repository may use different rules for formatting buffers. Therefore, Oracle CDD/Repository uses version numbers with each buffer. A version number includes both major and minor version numbers.

The minor version number indicates that the buffer has been changed in a way that is compatible with previous versions of the buffer. The major version number indicates that a possibly incompatible change has been made.

The version number is expressed as a decimal number of the form X.Y, where X is the major version number and Y is the minor version number.

Table C–1 lists the buffer versions for the current version of Oracle CDD/Repository.

Table C–1 Buffer Version Numbers

Buffer	Version Number
Access control list	1.0
Dictionary query	1.0
Directory information	2.0
Edit string	1.0
Expression	1.0

C.1.2 Buffer Body

There can be many structures (for example, elements, directory entries, and messages) in a repository, and there are many buffer formats used to describe them.

The kind of repository structure you manipulate determines the buffer type you use. For example, use a message buffer if you are manipulating messages. Each buffer's description includes the buffer type information as well as detailed information about the buffer format Oracle CDD/Repository expects.

How you plan to manipulate the repository structure determines how you interpret the body of the buffer.

C.1.3 Block Terminator

Each buffer ends with CDD$K_EOC, a byte constant that has the value 8.

C.2 Buffer Types

Because there are a number of different structures you need to manipulate as you call Oracle CDD/Repository, there are a number of different buffers. Table C–2 lists the buffers and their purposes.

Table C–2 Purpose of Buffers

Buffer	Purpose
Access control list buffer	Defines, changes, or displays access control list entries for repository elements.
Dictionary query buffer	Provides a set of retrieval criteria based on the presence and values of particular attributes and relationships under a particular repository element.
Directory information buffer	Describes one or more directory entries in the Oracle CDD/Repository directory system.
Edit string buffer	Describes edit string parameters.
Expression buffer	Describes complex formulas and expressions.

An unsigned longword is used to store the value for buffer types. Table C–3 shows each buffer type, its tag, and its tag value.

Table C–3 Buffer Types and Tag Values

Type of Buffer	Literal Constant	Tag Value
Access control list	cdd$k_acl_dsc	2818058
Dictionary query	cdd$k_dictionary_within_query	2818051
Directory information	cdd$k_directory_info_dsc	2818050
Edit string	cdd$k_edit_string_dsc	2818055
Expression	cdd$k_expression_buf_dsc	2818146

Tag values for buffers are in SYS$LIB:CDDTAGS.*

C.3 Simple Literals

Simple literals provide a value for buffers.

Format

```
<literal_value> ::=
        CDD$K_LITERAL
            {  DSC$K_DTYPE_ADT <binary_date_time>                        |
               DSC$K_DTYPE_T            <word_length>     <byte_string> |
               CDD$K_DTYPE_UNSTRUCTURED <longword_length> <byte_string> |
               <fixed_point_type>       <scale> <fixed_point_value>     |
               <floating_point_type>    <floating_point_value>         }

<word_length>          ::= unsigned word

<longword_length>      ::= unsigned longword

<byte_string>          ::= Zero or more contiguous bytes.

<fixed_point_type>     ::= {  DSC$K_DTYPE_W |
                              DSC$K_DTYPE_L |
                              DSC$K_DTYPE_Q }

<scale>                ::= unsigned byte scale factor.

<fixed_point_value>    ::= an integer value for an attribute

<floating_point_type>  ::= {  DSC$K_DTYPE_F |
                              DSC$K_DTYPE_G }

<floating_point_value> ::= a real value for an attribute
```

Explanation of Fields

The syntax elements valid in a simple literal are explained in the order in which they appear in the syntax diagram.

- CDD$K_LITERAL

  A tag indicating that a simple literal follows.

- DSC$K_DTYPE_ADT

  An unsigned byte indicating that an OpenVMS standard date and time follows.

- binary_date_time

  An 8-byte field containing an OpenVMS standard date and time.

- DSC$K_DTYPE_T

  An unsigned byte indicating that an ASCIW string follows. An ASCIW string is a string of bytes preceded by a 2-byte field containing the count of the bytes that follow.

- CDD$K_DTYPE_UNSTRUCTURED

  An unsigned byte indicating that an unstructured byte string follows.

- word_length

  A 2-byte field indicating the length in bytes of a string's value.

- longword_length

  A 4-byte field indicating the length in bytes of an unstructured value.

- byte_string

  Contiguous bytes containing the value of a text or unstructured attribute. There must be exactly as many bytes as indicated in the preceding length field.

- fixed_point_type

  A 1-byte field indicating one of three OpenVMS standard fixed-point data types: signed word (DSC$K_DTYPE_W), signed longword (DSC$K_ DTYPE_L), or signed quadword (DSC$K_DTYPE_Q).

- scale

  An unsigned byte field indicating a scale factor for a fixed-point field.

- floating_point_type

  A 1-byte field indicating one of two OpenVMS standard floating-point data types: F-floating (single precision) or G-floating (double precision).

Description

A simple literal can be included in other buffers; a simple literal of type CDD$K_UNSTRUCTURED can contain a complete edit-string or expression buffer.

C.4 Access Control List Buffer

Access control list buffers define, modify, and display access control list entries for repository elements. It is the value of the CDD$ACCESS_CONTROL_LIST attribute.

Format

```
<acl_buf> ::= <block_header>
              <body>
              <block_terminator>
```

Explanation of Fields

The syntax elements of the ACL buffer have the following meanings:

- block_header

  The header for an access control list buffer uses these values for the type and version numbers:

  ```
  <type>                ::= CDD$K_ACL_DSC
  <major_version_number> ::= 1
  <minor_version_number> ::= 0
  ```

- body

  Oracle CDD/Repository uses OpenVMS access control lists. Each privilege mask is a longword.

- block_terminator

  The unsigned longword tag CDD$K_EOC, marking the end of the access control list.

C.5 Dictionary Query Buffer

Dictionary query buffers describe within expressions. A **within expression** lets you navigate using relationships and attribute expressions in a repository from a starting point identified by a source element ID. A source element ID is specified by the source_elementid argument to the *MCS_scan_query* routine. (See Chapter 8 for more information.)

Format

```
<dictionary_within_query>  ::=     <block_header>
                                   <start_element_handle>
                                   <within_exp>
                                   <block_terminator>

<start_element_handle>     ::=      CDD$K_START_ELEMENT_HNDL <element_handle>

<within_exp>               ::=      CDD$K_WITHIN_EXP
                                  [ <query> ]
                                  [ <within_clause>... ]
                                    CDD$K_END

<within_clause>            ::=   { <relationship> [ <query> ] [ <target> ] |
                                    CDD$K_OWNING_RELATIONSHIP_ALL          |
                                    CDD$K_OWNED_BY_RELATIONSHIP_ALL         }

<relationship>             ::=
         { CDD$K_OWNING_RELATIONSHIP    | CDD$K_OWNED_BY_RELATIONSHIP      }
           <protocol_type> <version>

<target>                   ::=
         { CDD$K_ENTITY  | CDD$K_RELATIONSHIP  | CDD$K_ELEMENT }
         { <protocol_type> | CDD$K_ANY }
           <version>
         [ <query> ]

<query>                    ::= CDD$K_QUERY <within_condition>

<within_condition>         ::=
         { CDD$K_NOT <within_condition>                        |
           <boolean_op> <within_condition> <within_condition> |
           <relational_test>                                   |
           <existence_test>                                    |
           <uniqueness_tests>                                   }

<boolean_op>               ::= { CDD$K_AND | CDD$K_OR  | CDD$K_XOR }

<relational_test>          ::=
         { <relational_op>   <val_operand> <val_operand>      |
           CDD$K_EQL_ONE    <val_operand> <multi_val_operand> |
           CDD$K_ALPHABETIC <val_operand>                      }

<relational_op>             ::= {  CDD$K_EQL | CDD$K_NEQ | CDD$K_GTR |
                                   CDD$K_GEQ | CDD$K_LSS | CDD$K_LEQ }
```

```
<multi_val_operand>         ::=   <val_list> | <within_value>

<val_list>                  ::=    CDD$K_VALUE_LIST
                                   <val_operand> [...]
                                   CDD$K_END

<val_operand>               ::= { <attribute_protocol>   |
                                  <within_value>         |
                                  <simple_literal>        }

<within_value>              ::=   <within_exp> <attribute_protocol>

<existence_test>            ::=    CDD$K_EXISTS <exists_operand>

<exists_operand>            ::= { [ <within_exp> ] <attribute_protocol> |
                                    <within_exp>                        }

<uniqueness_tests>          ::=
    CDD$K_UNIQUE <within_exp> [ <attribute_protocol> ]

<attribute_protocol>   ::= CDD$K_ATTRIBUTE <protocol_type>
<version>              ::= <major_version_number> <minor_version_number>
<element_handle>       ::= unsigned quadword
<protocol_type>        ::= unsigned longword
<major_version_number> ::= unsigned longword
<minor_version_number> ::= unsigned longword
```

Explanation of Fields

The syntax elements valid in a dictionary query buffer are explained in the order in which they appear in the syntax diagram.

- block_header

  The header for a dictionary query buffer uses these values for the buffer type and version numbers:

  ```
  <type>                ::= CDD$K_DICTIONARY_WITHIN_QUERY
  <major_version_number> ::= 1
  <minor_version_number> ::= 0
  ```

- start_element_handle

  The handle of the dictionary element from which Oracle CDD/Repository begins interpreting the query. This element is called a source element ID or the *starting element* of the search. The element is specified by the source_ elementid argument to the *MCS_scan_query* routine. (See Chapter 8 for more information.)

- CDD$K_START_ELEMENT_HNDL

  An unsigned byte indicating that an element handle follows.

- element_handle

  An unsigned quadword containing the handle of the starting element. The handle must be valid for the session within which you pass the dictionary query buffer to an Oracle CDD/Repository entry point.

- within_exp

  The portion of the buffer that describes the relationships and elements you want Oracle CDD/Repository to return.

- CDD$K_WITHIN_EXP

  An unsigned byte marking the beginning of the within expression.

- query

  Selection criteria for elements or relationships you want Oracle CDD/Repository to return.

- within_clause

  A portion of the buffer indicating a relationship Oracle CDD/Repository follows.

- CDD$K_END

  An unsigned byte marking the end of the within expression.

- relationship

  A relationship for which you want information.

- target

  The target elements or relationships for Oracle CDD/Repository to return.

- CDD$K_OWNING_RELATIONSHIP_ALL

  An unsigned byte requesting all elements that own relationships of which the starting element is a member.

- CDD$K_OWNED_BY_RELATIONSHIP_ALL

  An unsigned byte requesting all elements that are members of relationships owned by the starting element.

- CDD$K_OWNING_RELATIONSHIP

  An unsigned byte requesting the owner of a particular relationship that owns the starting element.

- CDD$K_OWNED_BY_RELATIONSHIP

  An unsigned byte requesting the member of a relationship owned by the starting element.

- CDD$K_ENTITY

  An unsigned byte indicating the target element is an entity.

- CDD$K_RELATIONSHIP

  An unsigned byte indicating the target element is a relationship.

- CDD$K_ELEMENT

  An unsigned byte indicating the target element is either a relationship or an entity.

- protocol_type

  An unsigned byte identifying a valid Oracle CDD/Repository protocol type.

- CDD$K_ANY

  An unsigned byte indicating you want any item that fits the rest of the qualifications, no matter what its protocol type.

- CDD$K_QUERY

  An unsigned byte marking the beginning of a query.

- within_condition

  A Boolean expression Oracle CDD/Repository evaluates using the values of a repository element; the expression evaluates to true or false.

- CDD$K_NOT

  An unsigned byte specifying that the Boolean expression is considered true if the within condition is false.

- boolean_operator

  An unsigned byte identifying the Boolean operation you want to perform.

- CDD$K_AND

  An unsigned byte specifying that the Boolean expression is true only if both within conditions are true.

- CDD$K_OR

  An unsigned byte specifying that the Boolean expression is true if one or both of the within conditions are true.

- CDD$K_XOR

  An unsigned byte specifying that the Boolean expression is true only if one within condition is true and the other is false.

- relational_test

  An expression comprising an operator and one or more operands; it evaluates to true or false.

- relational_op

  An unsigned byte indicating how the values should be compared.

- CDD$K_EQL

  An unsigned byte specifying that the Boolean expression is true only if both value operands are equal.

- CDD$K_NEQ

  An unsigned byte specifying that the Boolean expression is true only if the value operands are not equal.

- CDD$K_GTR

  An unsigned byte specifying that the Boolean expression is true only if the first value operand exceeds the second.

- CDD$K_GEQ

  An unsigned byte specifying that the Boolean expression is true only if the first value operand equals or exceeds the second.

- CDD$K_LSS

  An unsigned byte specifying that the Boolean expression is true only if the second value operand exceeds the first.

- CDD$K_LEQ

  An unsigned byte specifying that the Boolean expression is true only if the second value operand equals or exceeds the first.

- val_operand

  A numeric or text value. Typically, you use the value of an attribute or a literal value here.

- CDD$K_EQL_ONE

  An unsigned byte specifying that the expression return true if the value operand is equal to one of the items in the multi_val_operand.

- CDD$K_ALPHABETIC

  An unsigned byte requesting that the relational test evaluate to true if the val_operand contains only alphabetic characters.

- multi_val_operand

  A stream of values for CDD$K_EQL_ONE to evaluate; is either a value list or a stream of elements given by a within expression.

- val_list

  A list of discrete values you list explicitly.

- within_value

  A stream of elements to be evaluated.

- CDD$K_VALUE_LIST

  An unsigned byte marking the beginning of a value list.

- simple_literal

  A simple literal expression. (See Section C.3.)

- existence_test

  A Boolean expression that evaluates to true only if the operand you supply exists in the repository.

- CDD$K_EXISTS

  An unsigned byte marking the beginning of the existence test.

- exists_operand

  An entity, attribute, or relationship Oracle CDD/Repository searches for using a within expression.

- uniqueness_tests

  A Boolean expression you use to ensure that there is exactly one entity, attribute, or relationship that meets the selection criteria you supply as an operand.

- CDD$K_UNIQUE

  An unsigned byte marking the beginning of the uniqueness test.

- attribute_protocol

  A portion of the buffer indicating the attribute you want.

- CDD$K_ATTRIBUTE

  An unsigned byte indicating that an attribute tag follows.

- version

  A pair of unsigned longwords indicating the major and minor versions of the buffer, element, or attribute that follows.

- block_terminator

  The unsigned byte CDD$K_EOC, indicating the end of the dictionary query buffer.

Description

If you omit the target clause, Oracle CDD/Repository returns handles for the relationships you request.

If you include the target clause, Oracle CDD/Repository returns handles of all the elements that participate in the relationship. The handles identify either the owner elements or the member elements, depending on whether you specified
CDD$K_OWNING_RELATIONSHIP or CDD$K_OWNED_BY_RELATIONSHIP.

You can use dictionary query buffers to find the following:

- descendants of an entity for which you already have a handle
- components of a data aggregate
- owner of an element for which you already have a handle

C.5.1 All Descendants

The following listing shows a buffer you use to return all descendants of a particular entity.

```
cdd$k_begin
cdd$k_dictionary_within_query 1 0
    cdd$k_within_exp
    cdd$k_owning_relationship_all
    cdd$k_end
cdd$k_eoc
```

C.5.2 Specific Descendants

The following listing shows a buffer you use to return all grandchildren in the second position of the current entity. It contains no start_element_handle.

```
cdd$k_begin
cdd$k_dictionary_within_query 1 0
    cdd$k_within_exp
    cdd$k_owning_relationship cdd$k_rel_da_contains 1 0
    cdd$k_owning_relationship cdd$k_rel_da_contains 1 0
    cdd$k_query
        cdd$k_eql
            cdd$k_attribute cdd$k_att_seq_number
        cdd$k_literal dsc$k_dtype_w 0 "2"
        cdd$k_entity cdd$k_ent_data_element 1 0
    cdd$k_end
cdd$k_eoc
```

C.5.3 Components of a Data Aggregate

The following listing shows a buffer you use to return all data elements of an aggregate given a particular element that is a member of that data aggregate.

```
cdd$k_begin
cdd$k_dictionary_within_query 1 0
    cdd$k_within_exp
    cdd$k_owned_by_relationship cdd$k_rel_da_contains 1 0
    cdd$k_owning_relationship cdd$k_rel_da_contains 1 0
        cdd$k_entity cdd$k_ent_data_element 1 0
    cdd$k_end
cdd$k_eoc
```

C.5.4 Element Owner

The following listing shows a buffer you use to find the field named FIELD_1 owned by a starting element identified by an element handle.

```
cdd$k_begin
cdd$k_dictionary_within_query 1 0
    cdd$k_start_element_hndl 12345
    cdd$k_within_exp
    cdd$k_owning_relationship cdd$k_rel_da_contains 1 0
        cdd$k_entity cdd$k_ent_data_element 1 0
        cdd$k_query
        cdd$k_eql
            cdd$k_attribute cdd$k_att_processing_name
            cdd$k_literal dsc$k_dtype_t 7 "FIELD_1"
    cdd$k_end
cdd$k_eoc
```

C.6 Directory Information Buffer

Directory information buffers describe an entry in an Oracle CDD/Repository system. You use this buffer to do the following:

- Define a directory entry for a repository element.

  To define a directory entry when you create an element, embed a directory information buffer in the metadata buffer you pass to the CDD$DEFINE_ELEMENT routine.

  To define a directory entry for an element that already exists in the repository, pass an element handle and a directory information buffer to the CDD$DEFINE_DIRECTORY_ENTRY routine.

- Retrieve one or more directory entries based on name or type.

  Pass a directory information buffer with the directory names of the elements for which you want directory information to the CDD$FETCH_START routine. Call the CDD$FETCH_NEXT routine to retrieve the next element that matches your selection criteria.

  For each element Oracle CDD/Repository finds, it returns a directory information buffer with the element's directory name and other information you requested, along with an element handle identifying that element. You can use this element handle to request further information about the element from an Oracle CDD/Repository system.

- Retrieve directory information for a single element whose element handle you already have.

  Embed a directory information buffer in a metadata buffer and pass it to the CDD$GET_ELEMENT routine, along with the element handle of the element for which you want information.

- Translate names according to Oracle CDD/Repository translation rules.

  Pass a buffer of directory names to CDD$TRANSLATE, which returns a buffer of fully translated names (see Section C.6).

Format

```
<directory_info>    ::=    <block_header>
                       [: <directory_name_list> |
                          <protocol_type>       |
                          <size>                |
                          <protocol_name>       |
                          <dictionary_type>     | :]
                          <block_terminator>
```

```
<directory_name_list> ::=    CDD$K_DIRECTORY_NAME_LIST
                             <name_spec> [...]
                             CDD$K_END

<name_spec>           ::=
    <directory_name> [ <default_name> ] [ <related_name> ]

<directory_name>      ::= CDD$K_DIRECTORY_NAME <dir_name>
<default_name>        ::= CDD$K_DEFAULT_NAME   <dir_name>
<related_name>        ::= CDD$K_RELATED_NAME   <dir_name>
<protocol_type>       ::= CDD$K_TYPE <type>
<size>                ::= CDD$K_SIZE <longword_length>
<protocol_name>       ::= CDD$K_PROTOCOL_NAME <word_length><byte_string>
<dictionary_type>     ::= CDD$K_DICTIONARY_TYPE [ CDD$K_CDD | CDD$K_NAD ]
<dir_name>            ::= <word_length><byte_string>
<type>                ::= unsigned longword
<size>                ::= unsigned longword
<longword_length>     ::= unsigned word
<word_length>         ::= unsigned word
<byte_string>         ::= characters
```

Explanation of Fields

The syntax elements valid in a directory information buffer are explained in the order in which they appear in the syntax diagram.

- block_header

  The header for a directory information buffer uses these values for buffer type and version number:

  ```
  <type>                ::= CDD$K_DIRECTORY_INFO_DSC
  <major_version_number> ::= 2
  <minor_version_number> ::= 0
  ```

- directory_name_list

  A portion of the buffer describing one or more directory entries by name.

- CDD$K_DIRECTORY_NAME_LIST

  An unsigned longword marking the beginning of the directory name list.

- name_spec

  The names of the element or set of elements you want. The name_spec includes the anchor (file system directory) and Oracle CDD/Repository path name and version number of the directory entry.

- directory_name

  A portion of the buffer identifying the anchor and Oracle CDD/Repository path name and version number of the directory entry. If you exclude parts of the directory name, Oracle CDD/Repository uses the value of the default_name field. Directory names can include logical names, search lists, and/or wildcard characters.

- CDD$K_DIRECTORY_NAME

  An unsigned byte indicating that a directory name follows.

- default_name

  A portion of the buffer that supplies values for portions of the directory name you do not supply.

- CDD$K_DEFAULT_NAME

  An unsigned byte indicating that the directory name that follows is a default name.

- related_name

  A portion of the buffer that supplies values for portions of the directory_ name. In general, use the related name to add a version number to the path names in the directory name list.

- CDD$K_RELATED_NAME

  An unsigned byte indicating that a directory name identifying a related name follows.

- dir_name

  A counted ASCII string giving the element's directory name. Oracle CDD/Repository interprets and translates the name according to the rules in the *Oracle CDD/Repository Architecture Manual*, which also explains the syntax of a path name.

- protocol_type

  The tag value of the entity this directory entry describes. Values for protocol type tags can be found in the tables in Appendix E.

- CDD$K_TYPE

  An unsigned byte indicating that the following longword identifies the type of the target element.

- type

  An unsigned longword identifying the target element's repository type.

- size

  A signed longword giving the approximate size of the directory entry. You cannot include this field in input buffers. Oracle CDD/Repository uses it on output buffers only.

- CDD$K_SIZE

  An unsigned byte indicating that the longword that follows contains the approximate size of the target element.

- longword_length

  An unsigned longword containing the approximate length of the target element.

- protocol_name

  The name of the protocol to which the repository element adheres. This parameter appears in output buffers only.

- CDD$K_PROTOCOL_NAME

- dictionary_type

  The format of the repository from which you want to retrieve directory information.

- CDD$K_DICTIONARY_TYPE

  Specify a repository type only for output buffers.

- CDD$K_CDD

  An unsigned byte indicating you want to retrieve information from DMU format dictionaries (dictionaries created by versions of Oracle CDD/Repository prior to Version 4.0).

- CDD$K_NAD

  An unsigned byte indicating you want to retrieve information from CDO format dictionaries (dictionaries created by Version 4.0 or higher.)

- CDD$K_END

  An unsigned byte marking the end of a directory name list.

- block_terminator

  The unsigned byte CDD$K_EOC, indicating the end of the directory information buffer.

Description

Oracle CDD/Repository generates a list of fully qualified path names (including anchor, path, and version) when it encounters the name_spec portion of the directory information buffer by using the following steps:

- It translates the first element of the path in the directory_name portion of the buffer. For example, in the path A.B.C, A is the first directory name. Oracle CDD/Repository repeats this step with the resulting path (or paths, if the directory name translates to a logical name search list). It continues translating the resulting paths, always translating only the first directory name. Oracle CDD/Repository stops translating when it cannot translate any further.

  At this point Oracle CDD/Repository has a list of path names, which may or may not have anchors.

- For the path names that do not have anchors, Oracle CDD/Repository does the following:

  – Looks first for the default_name portion of the buffer. If it exists, Oracle CDD/Repository prefixes the default_name to the path names without anchors, then once again tries to translate the first directory of the path name, stopping only when it cannot translate any further.

  – If the default_name portion of the buffer is not present, Oracle CDD/Repository checks to see if the user has set a default directory by calling CDD$SET_ANCHOR. If the user has called CDD$SET_ANCHOR, Oracle CDD/Repository prefixes the default directory to the path names without anchors. Then Oracle CDD/Repository again tries to translate the first directory in the path name, stopping only when it cannot translate any further.

  If at this point there are still path names without anchors, Oracle CDD/Repository prefixes the translation of CDD$COMPATIBILITY to those path names. The result is that all path names have anchors.

- For path names without version numbers, Oracle CDD/Repository looks for a version number in the related_name field. If a version number is there, Oracle CDD/Repository places it at the end of the path name.

Example

The following listing shows a buffer you use to define a directory entry. You can use this buffer as an input parameter to the CDD$FETCH_START routine or the CDD$DEFINE_DIRECTORY_ENTRY routine, or embed it in a metadata buffer passed to the CDD$DEFINE_ELEMENT routine.

```
cdd$k_begin
cdd$k_directory_info_dsc 2 0                          ❶
    cdd$k_directory_name_list                         ❷
    cdd$k_directory_name 19 "CDD$TOP.A.FIELD_1;1"     ❸
    cdd$k_end                                         ❹
cdd$k_eoc                                             ❺
```

❶ the block header for the directory information buffer

❷ the beginning of the directory name list

❸ the directory name, including an anchor specifying that the entry will be created in the compatibility repository

❹ tag marking the end of the directory name list

❺ tag marking the end of the directory information buffer

C.7 Edit String Buffer

Edit string buffers describe an edit string. You can provide edit strings as templates for displaying or entering data at run time.

Each language processor interprets different edit string characters differently.

Format

Each edit string tag in the following format is an unsigned word.

```
<edit_string>   ::=     <block_header>
                        <edit_string_buff>
                        <block_terminator>

<edit_string_buff> ::=
                        { <editing_tag>                          |
                          <valued_editing_tag> <length> <value> |
                          <edit_repeator> } [...]
                          CDD$K_EDIT_STR_END

<length> ::=            (a word value)

<value> ::=             (series of bytes that make up value of tag)

<valued_editing_tag> ::= { CDD$K_EDIT_STR_FLOAT_0_REPLACE |
                           CDD$K_EDIT_STR_LITERAL |
                           CDD$K_EDIT_STR_MINUS_LITERAL }
```

```
<editing_tag>  ::=    { CDD$K_EDIT_STR_ALPHABETIC        |
                       CDD$K_EDIT_STR_AM_PM              |
                       CDD$K_EDIT_STR_ANY_CHAR           |
                       CDD$K_EDIT_STR_COMMA              |
                       CDD$K_EDIT_STR_DAY_NUMBER         |
                       CDD$K_EDIT_STR_DECIMAL_DIGIT      |
                       CDD$K_EDIT_STR_DECIMAL_POINT      |
                       CDD$K_EDIT_STR_ENCODED_MINUS      |
                       CDD$K_EDIT_STR_ENCODED_PLUS       |
                       CDD$K_EDIT_STR_ENCODED_SIGN       |
                       CDD$K_EDIT_STR_EXPONENT           |
                       CDD$K_EDIT_STR_FLOAT_CURRENCY     |
                       CDD$K_EDIT_STR_FLOATING_MINUS     |
                       CDD$K_EDIT_STR_FLOATING_PLUS      |
                       CDD$K_EDIT_STR_FLOATING_SIGN      |
                       CDD$K_EDIT_STR_FLOAT_BLANK_SUPR   |
                       CDD$K_EDIT_STR_FRACTION_SECOND    |
                       CDD$K_EDIT_STR_HEX_DIGIT          |
                       CDD$K_EDIT_STR_HOUR_12            |
                       CDD$K_EDIT_STR_HOUR_24            |
                       CDD$K_EDIT_STR_JULIAN_DIGIT       |
                       CDD$K_EDIT_STR_LOGICAL_CHAR       |
                       CDD$K_EDIT_STR_LONG_TEXT          |
                       CDD$K_EDIT_STR_LOWERCASE          |
                       CDD$K_EDIT_STR_MINUS_PAREN        |
                       CDD$K_EDIT_STR_MINUTE             |
                       CDD$K_EDIT_STR_MONTH_NAME         |
                       CDD$K_EDIT_STR_MONTH_NUMBER       |
                       CDD$K_EDIT_STR_OCTAL_DIGIT        |
                       CDD$K_EDIT_STR_SECOND             |
                       CDD$K_EDIT_STR_MISSING_SEPARATOR  |
                       CDD$K_EDIT_STR_UPPERCASE          |
                       CDD$K_EDIT_STR_WEEKDAY_NAME       |
                       CDD$K_EDIT_STR_YEAR }

<edit_repeator> ::=    CDD$K_EDIT_STR_REPEATOR
                         CDD$K_EDIT_STR_REPEAT_COUNT
                           <wordcount>
                           { <editing_tag> |
                             <valued_editing_tag> <length> <value> }
                       CDD$K_EDIT_STR_END

<wordcount> ::=    unsigned word
```

Explanation of Fields

The fields of the edit string buffer have the following meanings:

- block_header

Uses the following values for the buffer type and version numbers:

```
<type>                ::= CDD$K_EDIT_STRING_DSC
<major_version_number> ::= 1
<minor_version_number> ::= 0
```

- CDD$K_EDIT_STR_ALPHABETIC

  Moves the next character in the field to the edited string. The character should be alphabetic. If the character is not alphabetic, the action to be taken depends on the rules of the language interpreting the string.

  For example:

```
CDD$K_BEGIN
    CDD$K_EDIT_STRING_DSC 1 0
        CDD$K_EDIT_STR_ALPHABETIC
        CDD$K_EDIT_STR_ALPHABETIC
        CDD$K_EDIT_STR_ALPHABETIC
        CDD$K_EDIT_STR_ALPHABETIC
        CDD$K_EDIT_STR_END
CDD$K_EOC
```

  Field value: WXYZ

  Edited value: WXYZ

- CDD$K_EDIT_STR_AM_PM

  Moves one character from one of the strings AM or PM to the edited string. Usually you need two occurrences of the CDD$K_EDIT_STR_AM_PM tag to produce correct output; combine it with the CDD$K_EDIT_STR_HOUR_12 tag and place it at the end of the edit string.

  For example:

```
CDD$K_BEGIN
    CDD$K_EDIT_STRING_DSC 1 0
        CDD$K_EDIT_STR_HOUR_12
        CDD$K_EDIT_STR_HOUR_12
        CDD$K_EDIT_STR_LITERAL 1 ":"
        CDD$K_EDIT_STR_MINUTE
        CDD$K_EDIT_STR_MINUTE
        CDD$K_EDIT_STR_LITERAL 1 " "
        CDD$K_EDIT_STR_AM_PM
        CDD$K_EDIT_STR_AM_PM
        CDD$K_EDIT_STR_END
CDD$K_EOC
```

  Field value: 11:30 a.m.

  Edited value: 11:30 AM

- CDD$K_EDIT_STR_ANY_CHAR

Moves the next character (8 bits) in the field's value to the edited string.

For example:

```
CDD$K_BEGIN
    CDD$K_EDIT_STRING_DSC 1 0
        CDD$K_EDIT_STR_REPEATOR
  CDD$K_EDIT_STR_REPEAT_COUNT 11
  CDD$K_EDIT_STR_ANY_CHAR
        CDD$K_EDIT_STR_END
        CDD$K_EDIT_STR_END
CDD$K_EOC
```

Field value: fj32dj%^*I

Edited value: fj32dj%^*I

- CDD$K_EDIT_STR_COMMA

  In nonnumeric values, inserts a comma into the edited string.

  In numeric values, inserts a comma into the edited string or suppresses a leading zero in the edited string.

  For example:

```
CDD$K_BEGIN
    CDD$K_EDIT_STRING_DSC 1 0
        CDD$K_EDIT_STR_FLOAT_CURRENCY
        CDD$K_EDIT_STR_FLOAT_CURRENCY
        CDD$K_EDIT_STR_COMMA
        CDD$K_EDIT_STR_FLOAT_CURRENCY
        CDD$K_EDIT_STR_FLOAT_CURRENCY
        CDD$K_EDIT_STR_DECIMAL_DIGIT
        CDD$K_EDIT_STR_DECIMAL_POINT
        CDD$K_EDIT_STR_DECIMAL_DIGIT
        CDD$K_EDIT_STR_DECIMAL_DIGIT
        CDD$K_EDIT_STR_END
CDD$K_EOC
```

  Field values: 1234.56 and 12.34

  Edited value: $1,234.56 and $12.34

- CDD$K_EDIT_STR_DAY_NUMBER

  Moves a digit of the day within month into the edited value. Use two consecutive occurrences of this tag; otherwise, only part of a 2-digit day is displayed.

For example:

```
CDD$K_BEGIN
    CDD$K_EDIT_STRING_DSC 1 0
        CDD$K_EDIT_STR_MONTH_NUMBER
        CDD$K_EDIT_STR_MONTH_NUMBER
        CDD$K_EDIT_STR_LITERAL 1 "/"
        CDD$K_EDIT_STR_DAY_NUMBER
        CDD$K_EDIT_STR_DAY_NUMBER
        CDD$K_EDIT_STR_LITERAL 1 "/"
        CDD$K_EDIT_STR_YEAR
        CDD$K_EDIT_STR_YEAR
        CDD$K_EDIT_STR_END
CDD$K_EOC
```

Field value: May 4, '85

Edited value: 5/04/85

- CDD$K_EDIT_STR_DECIMAL_DIGIT

  Moves the next decimal digit from the field's value into the edited string.

  Do not use this tag within an edit string containing either the CDD$K_ EDIT_STR_HEX_DIGIT or CDD$K_EDIT_STR_OCTAL_DIGIT tag.

  For example:

```
CDD$K_BEGIN
    CDD$K_EDIT_STRING_DSC 1 0
        CDD$K_EDIT_STR_DECIMAL_DIGIT
        CDD$K_EDIT_STR_DECIMAL_DIGIT
        CDD$K_EDIT_STR_DECIMAL_DIGIT
        CDD$K_EDIT_STR_END
CDD$K_EOC
```

  Field value: 613

  Edited value: 613

- CDD$K_EDIT_STR_DECIMAL_POINT

  Inserts a period (.) into the edited string. You can use this tag only once within an edit string for numeric fields.

For example:

```
CDD$K_BEGIN
    CDD$K_EDIT_STRING_DSC 1 0
        CDD$K_EDIT_STR_DECIMAL_DIGIT
        CDD$K_EDIT_STR_DECIMAL_DIGIT
        CDD$K_EDIT_STR_DECIMAL_POINT
        CDD$K_EDIT_STR_DECIMAL_DIGIT
        CDD$K_EDIT_STR_DECIMAL_DIGIT
        CDD$K_EDIT_STR_END
CDD$K_EOC
```

Field value: 2813E-2

Edited value: 28.13

- CDD$K_EDIT_STR_ENCODED_MINUS

  If the field's value is negative, this tag overpunches the next digit with a minus sign (–), then moves it to the edit string. If the field's value is positive or zero, this tag moves the next digit into the edited string.

  Use this tag only at the beginning or end of a string. Do not use this tag within an edit string containing any other tag designating a sign.

  For example:

```
CDD$K_BEGIN
    CDD$K_EDIT_STRING_DSC 1 0
        CDD$K_EDIT_STR_ENCODED_MINUS
        CDD$K_EDIT_STR_DECIMAL_DIGIT
        CDD$K_EDIT_STR_DECIMAL_DIGIT
        CDD$K_EDIT_STR_END
CDD$K_EOC
```

  Field value: –456

  Edited value: M56

- CDD$K_EDIT_STR_ENCODED_PLUS

  If the field's value is positive, this tag overpunches the next digit with a plus sign (+), then moves it to the edit string. If the field's value is negative or zero, this tag moves the next digit into the edited string.

  Use this tag only at the beginning or end of a string. Do not use this tag within an edit string containing any other tag designating a sign.

For example:

```
CDD$K_BEGIN
    CDD$K_EDIT_STRING_DSC 1 0
        CDD$K_EDIT_STR_ENCODED_PLUS
        CDD$K_EDIT_STR_DECIMAL_DIGIT
        CDD$K_EDIT_STR_DECIMAL_DIGIT
        CDD$K_EDIT_STR_END
CDD$K_EOC
```

Field value: +123

Edited value: A23

- CDD$K_EDIT_STR_ENCODED_SIGN

  If the field's value is positive, this tag overpunches the next digit with a plus sign (+), then moves it to the edit string. If the field's value is negative, this tag overpunches the next digit with a minus sign (–), then moves it to the edit string.

  If the field's value is zero, the action of this tag depends on the language processor.

  Use this tag only at the beginning or end of a string. Do not use this tag within an edit string containing any other tag designating a sign.

  For example:

```
CDD$K_BEGIN
    CDD$K_EDIT_STRING_DSC 1 0
        CDD$K_EDIT_STR_ENCODED_SIGN
        CDD$K_EDIT_STR_DECIMAL_DIGIT
        CDD$K_EDIT_STR_DECIMAL_DIGIT
        CDD$K_EDIT_STR_END
CDD$K_EOC
```

  Field values: +123 & –456

  Edited values: A23 & M56

- CDD$K_EDIT_STR_EXPONENT

  Divides an edit string into two parts for floating-point or scientific notation. The first part is the mantissa and the second part is the exponent.

For example:

```
CDD$K_BEGIN
    CDD$K_EDIT_STRING_DSC 1 0
        CDD$K_EDIT_STR_FLOATING_SIGN
        CDD$K_EDIT_STR_DECIMAL_DIGIT
        CDD$K_EDIT_STR_DECIMAL_DIGIT
        CDD$K_EDIT_STR_EXPONENT
        CDD$K_EDIT_STR_FLOATING_SIGN
        CDD$K_EDIT_STR_DECIMAL_DIGIT
        CDD$K_EDIT_STR_DECIMAL_DIGIT
        CDD$K_EDIT_STR_END
CDD$K_EOC
```

Field value: 1200

Edited value: +12E+02

- CDD$K_EDIT_STR_FLOAT_CURRENCY

  Inserts the currency sign to the left of the first printed digit of the field's value. One occurrence of CDD$K_EDIT_STR_FLOAT_CURRENCY inserts the currency sign in the next character position in the edited string. If you supply this tag more than once at the beginning of an edit string, it suppresses leading zeros. To suppress up to four leading zeros, supply this tag four times at the beginning of the edit string.

  For example:

```
CDD$K_BEGIN
    CDD$K_EDIT_STRING_DSC 1 0
        CDD$K_EDIT_STR_FLOAT_CURRENCY
        CDD$K_EDIT_STR_COMMA
        CDD$K_EDIT_STR_FLOAT_CURRENCY
        CDD$K_EDIT_STR_FLOAT_CURRENCY
        CDD$K_EDIT_STR_FLOAT_CURRENCY
        CDD$K_EDIT_STR_DECIMAL_POINT
        CDD$K_EDIT_STR_LITERAL 1 "0"
        CDD$K_EDIT_STR_LITERAL 1 "0"
        CDD$K_EDIT_STR_END
CDD$K_EOC
```

  Field value: 157.86

  Edited value: $158.00

- CDD$K_EDIT_STR_FLOATING_MINUS

  Inserts a minus sign (–) in the edited field.

  If the field is nonnumeric, this tag moves a minus sign into the edited field.

If the field is numeric, a single occurrence of this tag inserts a blank into the edited string when the field's value is positive and a minus sign when the field's value is negative. If the field's value is zero, the effect of this tag depends on the language processor reading the field. Several occurrences of this tag at the beginning of an edit string suppresses leading zeros before inserting a blank or minus sign into the edited string.

Do not use this tag in the same edit string as any other tag designating a sign.

The following edit string suppresses three leading zeros and places a minus sign in front of the edited string:

```
CDD$K_BEGIN
    CDD$K_EDIT_STRING_DSC 1 0
        CDD$K_EDIT_STR_FLOATING_MINUS
        CDD$K_EDIT_STR_FLOATING_MINUS
        CDD$K_EDIT_STR_FLOATING_MINUS
        CDD$K_EDIT_STR_DECIMAL_DIGIT
        CDD$K_EDIT_STR_END
CDD$K_EOC
```

Field value: –54

Edited value: –54

- CDD$K_EDIT_STR_FLOATING_PLUS

  Inserts a plus sign (+) in the edited field.

  If the field is nonnumeric, this tag moves a plus sign into the edited field.

  A single occurrence of the CDD$K_EDIT_STR_FLOATING_PLUS tag inserts a blank into the edited string when the field's value is negative and a plus sign when the field's value is positive. If the field's value is zero, the effect of this tag depends on the language processor reading the field.

  More than one occurrence of this tag at the beginning of an edit string suppresses leading zeros and inserts a blank or plus sign into the edited string.

  You cannot use this tag in an edit string that uses another tag to designate a sign.

For example:

```
CDD$K_BEGIN
    CDD$K_EDIT_STRING_DSC 1 0
        CDD$K_EDIT_STR_FLOATING_PLUS
        CDD$K_EDIT_STR_FLOATING_PLUS
        CDD$K_EDIT_STR_FLOATING_PLUS
        CDD$K_EDIT_STR_DECIMAL_DIGIT
        CDD$K_EDIT_STR_END
CDD$K_EOC
```

Field value: 54

Edited value: +54

- CDD$K_EDIT_STR_FLOATING_SIGN

  Inserts a minus sign (–) or plus sign (+) in the edited field.

  If the field is nonnumeric, this tag moves a minus sign into the edited field.

  A single occurrence of the CDD$K_EDIT_STR_FLOATING_SIGN tag inserts a plus sign into the edited string when the field's value is positive and a minus sign when the field's value is negative. If the field's value is zero, the effect of this tag depends on the language processor reading the field.

  If you use this tag a number of times at the beginning of an edit string, the tag suppresses leading zeros before inserting a plus sign or minus sign into the edited string.

  You cannot use this tag in an edit string that uses another tag to designate a sign.

  For example:

```
CDD$K_BEGIN
    CDD$K_EDIT_STRING_DSC 1 0
        CDD$K_EDIT_STR_FLOATING_SIGN
        CDD$K_EDIT_STR_FLOATING_SIGN
        CDD$K_EDIT_STR_FLOATING_SIGN
        CDD$K_EDIT_STR_DECIMAL_DIGIT
        CDD$K_EDIT_STR_END
CDD$K_EOC
```

  Field value: 54

  Edited value: +54

- CDD$K_EDIT_STR_FLOAT_BLANK_SUPR

  Suppresses blanks from an edited field. If the value of the character is a blank, the edited field excludes it.

For example:

```
CDD$K_BEGIN
    CDD$K_EDIT_STRING_DSC 1 0
        CDD$K_EDIT_STR_MONTH_NAME
        CDD$K_EDIT_STR_MONTH_NAME
        CDD$K_EDIT_STR_MONTH_NAME
        CDD$K_EDIT_STR_FLOAT_BLANK_SUPR
        CDD$K_EDIT_STR_FLOAT_BLANK_SUPR
        CDD$K_EDIT_STR_FLOAT_BLANK_SUPR
        CDD$K_EDIT_STR_FLOAT_BLANK_SUPR
        CDD$K_EDIT_STR_FLOAT_BLANK_SUPR
        CDD$K_EDIT_STR_COMMA
        CDD$K_EDIT_STR_YEAR
        CDD$K_EDIT_STR_YEAR
        CDD$K_EDIT_STR_YEAR
        CDD$K_EDIT_STR_YEAR
        CDD$K_EDIT_STR_END
CDD$K_EOC
```

Field value: June 15, 1982

Edited value: JUNE,1982

- CDD$K_EDIT_STR_FLOAT_0_REPLACE

  If the next digit of the field's value is zero, this tag moves the specified literal value into the edited string. Otherwise, this tag displays the digit.

  For example:

```
CDD$K_BEGIN
    CDD$K_EDIT_STRING_DSC 1 0
        CDD$K_EDIT_STR_FLOAT_0_REPLACE 1 " "
        CDD$K_EDIT_STR_DECIMAL_DIGIT
        CDD$K_EDIT_STR_DECIMAL_DIGIT
        CDD$K_EDIT_STR_END
CDD$K_EOC
```

  Field value: 25

  Edited value: #25

```
CDD$K_BEGIN
    CDD$K_EDIT_STRING_DSC 1 0
        CDD$K_EDIT_STR_FLOAT_0_REPLACE 1 "*"
        CDD$K_EDIT_STR_DECIMAL_DIGIT
        CDD$K_EDIT_STR_DECIMAL_DIGIT
        CDD$K_EDIT_STR_END
CDD$K_EOC
```

  Field value: 25

  Edited value: *25

Table C–4 shows how the COBOL, DATATRIEVE, PL/I, and RPG language processors interpret the CDD$K_EDIT_STR_FLOAT_0_REPLACE tag.

Table C–4 Translation of CDD$K_EDIT_STR_FLOAT_0_REPLACE Characters

Z String Format	Character in String	COBOL PICTURE	DTR EDIT	PL/I PICTURE	RPG EDIT WORD
Z"string"	blank	Z	Z	Z or Y[1]	0
	*	*	*	*	*
	any other character	*	*	*	*

[1]PL/I sometimes translates a blank to Z and sometimes to Y. Refer to PL/I documentation for more information.

- CDD$K_EDIT_STR_FRACTION_SECOND

  Moves a value for fractions of a second within a time field into the edited value. Use this tag twice consecutively to denote hundredths of a second.

  For example:

```
CDD$K_BEGIN
    CDD$K_EDIT_STRING_DSC 1 0
        CDD$K_EDIT_STR_MINUTE
        CDD$K_EDIT_STR_MINUTE
        CDD$K_EDIT_STR_LITERAL 1 ":"
        CDD$K_EDIT_STR_SECOND
        CDD$K_EDIT_STR_SECOND
        CDD$K_EDIT_STR_LITERAL 1 "."
        CDD$K_EDIT_STR_FRACTION_SECOND
        CDD$K_EDIT_STR_FRACTION_SECOND
        CDD$K_EDIT_STR_END
CDD$K_EOC
```

  Field value: 23 minutes 13.56 seconds

  Edited value: 23:13.56

- CDD$K_EDIT_STR_HEX_DIGIT

  Moves a hexadecimal digit from the field's value into the edited string.

  Do not use this tag within an edit string containing either the CDD$K_EDIT_STR_DECIMAL_DIGIT or CDD$K_EDIT_STR_OCTAL_DIGIT tags.

For example:

```
CDD$K_BEGIN
    CDD$K_EDIT_STRING_DSC 1 0
        CDD$K_EDIT_STR_HEX_DIGIT
        CDD$K_EDIT_STR_HEX_DIGIT
        CDD$K_EDIT_STR_HEX_DIGIT
        CDD$K_EDIT_STR_END
CDD$K_EOC
```

Field value: 32

Edited value: 020

- CDD$K_EDIT_STR_HOUR_12

  Moves one digit of the hour, in 12-hour mode, into the edited value. You should use two consecutive instances of this tag.

  Do not use the CDD$K_EDIT_STR_HOUR_12 tag in the same edit string with the CDD$K_EDIT_STR_HOUR_24 tag.

  For example:

```
CDD$K_BEGIN
    CDD$K_EDIT_STRING_DSC 1 0
        CDD$K_EDIT_STR_HOUR_12
        CDD$K_EDIT_STR_HOUR_12
        CDD$K_EDIT_STR_LITERAL 1 ":"
        CDD$K_EDIT_STR_MINUTE
        CDD$K_EDIT_STR_MINUTE
        CDD$K_EDIT_STR_LITERAL 1 " "
        CDD$K_EDIT_STR_AM_PM
        CDD$K_EDIT_STR_AM_PM
        CDD$K_EDIT_STR_END
CDD$K_EOC
```

  Field value: 11:30 a.m.

  Edited value: 11:30 AM

- CDD$K_EDIT_STR_HOUR_24

  Moves one digit of the hour, in 24-hour mode, into the edited value. You should use two consecutive instances of this tag.

  Do not use the CDD$K_EDIT_STR_HOUR_24 tag in the same edit string with a CDD$K_EDIT_STR_HOUR_12 tag.

For example:

```
CDD$K_BEGIN
    CDD$K_EDIT_STRING_DSC 1 0
        CDD$K_EDIT_STR_HOUR_24
        CDD$K_EDIT_STR_HOUR_24
        CDD$K_EDIT_STR_LITERAL 1 ":"
        CDD$K_EDIT_STR_MINUTE
        CDD$K_EDIT_STR_MINUTE
        CDD$K_EDIT_STR_END
CDD$K_EOC
```

Field value: 2:30 p.m.

Edited value: 14:30

- CDD$K_EDIT_STR_JULIAN_DIGIT

  Moves a digit of the Julian date into the edited value.

  For example:

```
CDD$K_BEGIN
    CDD$K_EDIT_STRING_DSC 1 0
        CDD$K_EDIT_STR_YEAR
        CDD$K_EDIT_STR_YEAR
        CDD$K_EDIT_STR_YEAR
        CDD$K_EDIT_STR_YEAR
        CDD$K_EDIT_STR_LITERAL 1 "/"
        CDD$K_EDIT_STR_JULIAN_DIGIT
        CDD$K_EDIT_STR_JULIAN_DIGIT
        CDD$K_EDIT_STR_JULIAN_DIGIT
        CDD$K_EDIT_STR_END
CDD$K_EOC
```

  Field value: 6/4/80

  Edited value: 1980/156

- CDD$K_EDIT_STR_LITERAL

  Inserts a character string into the edited value.

  For example:

```
CDD$K_BEGIN
    CDD$K_EDIT_STRING_DSC 1 0
        CDD$K_EDIT_STR_DECIMAL_DIGIT
        CDD$K_EDIT_STR_DECIMAL_DIGIT
        CDD$K_EDIT_STR_LITERAL 1 " "
        CDD$K_EDIT_STR_LITERAL 5 "Hours"
        CDD$K_EDIT_STR_END
CDD$K_EOC
```

  Field value: 40

Edited value: 40 Hours

For example:

```
CDD$K_BEGIN
    CDD$K_EDIT_STRING_DSC 1 0
        CDD$K_EDIT_STR_FLOAT_CURRENCY
        CDD$K_EDIT_STR_COMMA
        CDD$K_EDIT_STR_FLOAT_CURRENCY
        CDD$K_EDIT_STR_FLOAT_CURRENCY
        CDD$K_EDIT_STR_FLOAT_CURRENCY
        CDD$K_EDIT_STR_DECIMAL_POINT
        CDD$K_EDIT_STR_LITERAL 2 "00"
        CDD$K_EDIT_STR_END
CDD$K_EOC
```

Field value: 157.86

Edited value: $158.00

Table C–5 shows how the CDO, COBOL, DATATRIEVE, PL/I, and RPG language processors translate literal edit strings for their own use. In the table, <i> means that the language does not support the character; the language ignores the character.

If an edit string consists of only a blank, zero (0), slash (/), or percent sign (%), the language processor translates the string as shown in the first four lines of the table. The last line of the table shows how the language processor translates any other edit string.

For example, DATATRIEVE translates the edit string "sample string" to 'sample string'.

Table C–5 Translation of CDD$K_EDIT_STR_LITERAL Edit Strings

Literal String Format	CDO String	COBOL PICTURE	DTR EDIT	PL/I PICTURE	RPG EDIT WORD
"string"	blank	B	B	B	&
	0	0	0	<i>	0
	/	/	/	/	/
	%	<i>	%	<i>	%
	any other string	<i>	'sample string'	<i>	string

- CDD$K_EDIT_STR_LOGICAL_CHAR

  Displays logical fields as TRUE or FALSE.

  For example:

```
CDD$K_BEGIN
    CDD$K_EDIT_STRING_DSC 1 0
        CDD$K_EDIT_STR_LOGICAL_CHAR
        CDD$K_EDIT_STR_LOGICAL_CHAR
        CDD$K_EDIT_STR_LOGICAL_CHAR
        CDD$K_EDIT_STR_LOGICAL_CHAR
        CDD$K_EDIT_STR_LOGICAL_CHAR
        CDD$K_EDIT_STR_END
CDD$K_EOC
```

  Field value: 0

  Edited value: FALSE

- CDD$K_EDIT_STR_LONG_TEXT

  Displays a long text string on separate lines. Use several consecutive instances of this tag; the number of instances you use is the number of characters printed per line.

  For example:

```
CDD$K_BEGIN
    CDD$K_EDIT_STRING_DSC 1 0
        CDD$K_EDIT_STR_LONG_TEXT
        CDD$K_EDIT_STR_LONG_TEXT
        CDD$K_EDIT_STR_LONG_TEXT
        CDD$K_EDIT_STR_LONG_TEXT
        CDD$K_EDIT_STR_LONG_TEXT
        CDD$K_EDIT_STR_END
CDD$K_EOC
```

  Field value: 1234567890

  Edited value:
```
12345
67890
```

- CDD$K_EDIT_STR_LOWERCASE

  Prints any remaining alphabetic characters in the field's value in lowercase.

For example:

```
CDD$K_BEGIN
    CDD$K_EDIT_STRING_DSC 1 0
        CDD$K_EDIT_STR_LOWERCASE
        CDD$K_EDIT_STR_MONTH_NAME
        CDD$K_EDIT_STR_MONTH_NAME
        CDD$K_EDIT_STR_MONTH_NAME
        CDD$K_EDIT_STR_LITERAL 1 " "
        CDD$K_EDIT_STR_DAY_NUMBER
        CDD$K_EDIT_STR_DAY_NUMBER
        CDD$K_EDIT_STR_END
CDD$K_EOC
```

Field value: November 12th

Edited value: nov 12

- CDD$K_EDIT_STR_MINUS_LITERAL

  Replaces a negative sign in a field's value with a literal you supply. If the field's value is positive, this tag moves a blank to the string instead of the literal you supply.

  Do not use this tag within an edit string containing any other tag designating a sign.

  For example:

```
CDD$K_BEGIN
    CDD$K_EDIT_STRING_DSC 1 0
        CDD$K_EDIT_STR_DECIMAL_DIGIT
        CDD$K_EDIT_STR_DECIMAL_DIGIT
        CDD$K_EDIT_STR_MINUS_LITERAL 2 "CR"
        CDD$K_EDIT_STR_END
CDD$K_EOC
```

```
CDD$K_BEGIN
    CDD$K_EDIT_STRING_DSC 1 0
        CDD$K_EDIT_STR_DECIMAL_DIGIT
        CDD$K_EDIT_STR_DECIMAL_DIGIT
        CDD$K_EDIT_STR_MINUS_LITERAL 2 "DB"
        CDD$K_EDIT_STR_END
CDD$K_EOC
```

  Field value: –15

  Edited value: 15CR & 15DB

  Table C–6 shows how the CDO, COBOL, DATATRIEVE, PL/I, and RPG language processors translate the CDD$K_EDIT_STR_MINUS_LITERAL tag.

Table C–6 Translation of CDD$K_EDIT_STR_MINUS_LITERAL

Minus Literal String Format	CDO String	COBOL PICTURE	DTR EDIT	PL/I PICTURE	RPG EDIT WORD
&"string"	CR	CR	CR	CR	CR
	DB	DB	DB	DB	DB
	any other string	–	–	–	string

- CDD$K_EDIT_STR_MINUS_PAREN

  Encloses negative values in parentheses.

  Do not use this tag in the same edit string with any other tag designating a sign.

  For example:

```
CDD$K_BEGIN
    CDD$K_EDIT_STRING_DSC 1 0
        CDD$K_EDIT_STR_MINUS_PAREN
        CDD$K_EDIT_STR_DECIMAL_DIGIT
        CDD$K_EDIT_STR_DECIMAL_DIGIT
        CDD$K_EDIT_STR_DECIMAL_DIGIT
        CDD$K_EDIT_STR_END
CDD$K_EOC
```

  Field value: –678

  Edited value: (678)

- CDD$K_EDIT_STR_MINUTE

  Moves one digit for the number of minutes in a time value into the edit string. Use two consecutive instances of this tag to print both digits of a 2-digit number.

For example:

```
CDD$K_BEGIN
    CDD$K_EDIT_STRING_DSC 1 0
        CDD$K_EDIT_STR_HOUR_12
        CDD$K_EDIT_STR_HOUR_12
        CDD$K_EDIT_STR_LITERAL 1 ":"
        CDD$K_EDIT_STR_MINUTE
        CDD$K_EDIT_STR_MINUTE
        CDD$K_EDIT_STR_LITERAL 1 " "
        CDD$K_EDIT_STR_AM_PM
        CDD$K_EDIT_STR_AM_PM
        CDD$K_EDIT_STR_END
CDD$K_EOC
```

Field value: 22:30

Edited value: 10:30 PM

- CDD$K_EDIT_STR_MISSING_SEPARATOR

  If the field has a missing value attribute or relationship, this tag separates two edit strings. If the field value is not the missing value, the first edit string controls the output of the field. If the content of the field is the missing value, the second edit string controls the output of the field.

  For example:

```
CDD$K_BEGIN
    CDD$K_EDIT_STRING_DSC 1 0
        CDD$K_EDIT_STR_DECIMAL_DIGIT
        CDD$K_EDIT_STR_DECIMAL_DIGIT
        CDD$K_EDIT_STR_DECIMAL_DIGIT
        CDD$K_EDIT_STR_MISSING_SEPARATOR
        CDD$K_EDIT_STR_LITERAL 7 "Unknown"
        CDD$K_EDIT_STR_END
CDD$K_EOC
```

  Field value: missing value

  Edited value: Unknown

- CDD$K_EDIT_STR_MONTH_NAME

  Moves the next letter of the month name into the edited field. To avoid ambiguous abbreviations, use at least three occurrences of this tag.

For example:

```
CDD$K_BEGIN
    CDD$K_EDIT_STRING_DSC 1 0
        CDD$K_EDIT_STR_MONTH_NAME
        CDD$K_EDIT_STR_MONTH_NAME
        CDD$K_EDIT_STR_MONTH_NAME
        CDD$K_EDIT_STR_LITERAL 1 " "
        CDD$K_EDIT_STR_DAY_NUMBER
        CDD$K_EDIT_STR_DAY_NUMBER
        CDD$K_EDIT_STR_LITERAL 1 " "
        CDD$K_EDIT_STR_YEAR
        CDD$K_EDIT_STR_YEAR
        CDD$K_EDIT_STR_END
CDD$K_EOC
```

Field value: September 4, '88

Edited value: SEP 4 88

- CDD$K_EDIT_STR_MONTH_NUMBER

  Moves a digit for the number of the month to the edited field. You should use two consecutive instances of this tag in any edit string.

  For example:

```
CDD$K_BEGIN
    CDD$K_EDIT_STRING_DSC 1 0
        CDD$K_EDIT_STR_MONTH_NUMBER
        CDD$K_EDIT_STR_MONTH_NUMBER
        CDD$K_EDIT_STR_LITERAL 1 "/"
        CDD$K_EDIT_STR_DAY_NUMBER
        CDD$K_EDIT_STR_DAY_NUMBER
        CDD$K_EDIT_STR_LITERAL 1 "/"
        CDD$K_EDIT_STR_YEAR
        CDD$K_EDIT_STR_YEAR
        CDD$K_EDIT_STR_END
CDD$K_EOC
```

  Field value: May 4, '88

  Edited value: 5/04/88

- CDD$K_EDIT_STR_OCTAL_DIGIT

  Moves an octal digit from the field's value to the edited string.

  Do not use this tag in edit strings containing either the CDD$K_EDIT_STR_HEX_DIGIT or CDD$K_EDIT_STR_DECIMAL_DIGIT tags.

For example:

```
CDD$K_BEGIN
    CDD$K_EDIT_STRING_DSC 1 0
        CDD$K_EDIT_STR_OCTAL_DIGIT
        CDD$K_EDIT_STR_OCTAL_DIGIT
        CDD$K_EDIT_STR_OCTAL_DIGIT
        CDD$K_EDIT_STR_END
CDD$K_EOC
```

Field value: 32

Edited value: 040

- CDD$K_EDIT_STR_REPEATOR

  Defines a repeating edit string character and the number of times it repeats.

- CDD$K_EDIT_STR_REPEAT_COUNT

  Indicates the number of times the tag you supply is repeated within an edit string.

  For example:

```
CDD$K_BEGIN
    CDD$K_EDIT_STRING_DSC 1 0
        CDD$K_EDIT_STR_REPEATOR
        CDD$K_EDIT_STR_REPEAT_COUNT 9
        CDD$K_EDIT_STR_WEEKDAY_NAME
        CDD$K_EDIT_STR_END
        CDD$K_EDIT_STR_END
CDD$K_EOC
```

Field value: June 4, 1980

Edited value: Wednesday

- CDD$K_EDIT_STR_SECOND

  Moves a digit for the number of seconds in a time value into the edited string. You should use two consecutive instances of this tag in any edit string.

For example:

```
CDD$K_BEGIN
    CDD$K_EDIT_STRING_DSC 1 0
        CDD$K_EDIT_STR_MINUTE
        CDD$K_EDIT_STR_MINUTE
        CDD$K_EDIT_STR_LITERAL 1 ":"
        CDD$K_EDIT_STR_SECOND
        CDD$K_EDIT_STR_SECOND
        CDD$K_EDIT_STR_LITERAL 1 "."
        CDD$K_EDIT_STR_FRACTION_SECOND
        CDD$K_EDIT_STR_FRACTION_SECOND
        CDD$K_EDIT_STR_END
CDD$K_EOC
```

Field value: 23 minutes 13.56 seconds

Edited value: 23:13.56

- CDD$K_EDIT_STR_UPPERCASE

  Prints any remaining alphabetic characters in the field's value in uppercase.

  For example:

```
CDD$K_BEGIN
    CDD$K_EDIT_STRING_DSC 1 0
        CDD$K_EDIT_STR_UPPERCASE
        CDD$K_EDIT_STR_REPEATOR
        CDD$K_EDIT_STR_REPEAT_COUNT 20
        CDD$K_EDIT_STR_ANY_CHAR
        CDD$K_EDIT_STR_END
        CDD$K_EDIT_STR_END
CDD$K_EOC
```

  Field value: Jones

  Edited value: JONES

- CDD$K_EDIT_STR_WEEKDAY_NAME

  Moves the next letter from the day of week in a time value into the edited value.

  For example:

```
CDD$K_BEGIN
    CDD$K_EDIT_STRING_DSC 1 0
        CDD$K_EDIT_STR_REPEATOR
            CDD$K_EDIT_STR_REPEAT_COUNT 1 9
            CDD$K_EDIT_STR_WEEKDAY_NAME
        CDD$K_EDIT_STR_END
        CDD$K_EDIT_STR_END
CDD$K_EOC
```

Field value: June 4, 1980

Edited value: Wednesday

- CDD$K_EDIT_STR_YEAR

  Moves the next digit of the year in a time value into the edited string.

  For example:

```
CDD$K_BEGIN
    CDD$K_EDIT_STRING_DSC 1 0
        CDD$K_EDIT_STR_MONTH_NAME
        CDD$K_EDIT_STR_MONTH_NAME
        CDD$K_EDIT_STR_MONTH_NAME
        CDD$K_EDIT_STR_LITERAL 1 " "
        CDD$K_EDIT_STR_DAY_NUMBER
        CDD$K_EDIT_STR_DAY_NUMBER
        CDD$K_EDIT_STR_LITERAL 1 " "
        CDD$K_EDIT_STR_YEAR
        CDD$K_EDIT_STR_YEAR
        CDD$K_EDIT_STR_END
CDD$K_EOC
```

Field value: May 4, '80

Edited value: MAY 4 80

Example

```
cdd$k_begin
cdd$k_metadata_buf_dsc 1 5
    cdd$k_entity cdd$k_ent_data_element 1 0
    cdd$k_begin
    cdd$k_directory_info_dsc 2 0
        cdd$k_directory_name_list
        cdd$k_directory_name 8 "CURRENCY"
        cdd$k_end
        cdd$k_type cdd$k_ent_data_element
    cdd$k_eoc
```

```
        cdd$k_attribute_list
        cdd$k_attribute cdd$k_att_processing_name
        cdd$k_literal dsc$k_dtype_t 8 "CURRENCY"
        cdd$k_attribute cdd$k_att_de_datatype
        cdd$k_literal dsc$k_dtype_w 0 dsc$k_dtype_f
        cdd$k_attribute cdd$k_att_de_edit_string
        cdd$k_literal cdd$k_dtype_unstructured 34
    cdd$k_begin
        cdd$k_edit_string_dsc 1 0
            cdd$k_edit_str_float_currency
            cdd$k_edit_str_float_currency
            cdd$k_edit_str_comma
            cdd$k_edit_str_float_currency
            cdd$k_edit_str_float_currency
            cdd$k_edit_str_decimal_digit
            cdd$k_edit_str_decimal_point
            cdd$k_edit_str_decimal_digit
            cdd$k_edit_str_decimal_digit
            cdd$k_edit_str_end
            cdd$k_eoc
    cdd$k_end
cdd$k_eoc
```

This listing shows a complete edit string buffer, including buffer header and terminator, embedded in a metadata buffer.

C.7.1 Edit String Buffer Tags

Edit string buffer tags are unsigned words. Table C–7 shows the value of each edit string buffer tag.

Table C–7 Edit String Buffer Tags

Edit String Buffer Tag	Tag Value
cdd$k_edit_str_alphabetic	1
cdd$k_edit_str_am_pm	2
cdd$k_edit_str_any_char	3
cdd$k_edit_str_comma	4
cdd$k_edit_str_day_number	5
cdd$k_edit_str_decimal_digit	6
cdd$k_edit_str_decimal_point	7
cdd$k_edit_str_encoded_minus	8

(continued on next page)

Table C–7 (Cont.) Edit String Buffer Tags

Edit String Buffer Tag	Tag Value
cdd$k_edit_str_encoded_plus	9
cdd$k_edit_str_encoded_sign	10
cdd$k_edit_str_exponent	11
cdd$k_edit_str_float_currency	12
cdd$k_edit_str_floating_minus	13
cdd$k_edit_str_floating_plus	14
cdd$k_edit_str_floating_sign	15
cdd$k_edit_str_float_blank_supr	16
cdd$k_edit_str_float_0_replace	17
cdd$k_edit_str_fraction_second	18
cdd$k_edit_str_hex_digit	19
cdd$k_edit_str_hour_12	20
cdd$k_edit_str_hour_24	21
cdd$k_edit_str_julian_digit	22
cdd$k_edit_str_literal	23
cdd$k_edit_str_logical_char	24
cdd$k_edit_str_long_text	25
cdd$k_edit_str_lowercase	26
cdd$k_edit_str_minus_literal	27
cdd$k_edit_str_minus_paren	28
cdd$k_edit_str_minute	29
cdd$k_edit_str_missing_separator	30
cdd$k_edit_str_month_name	31
cdd$k_edit_str_month_number	32
cdd$k_edit_str_octal_digit	33
cdd$k_edit_str_repeator	34
cdd$k_edit_str_repeat_count	35
cdd$k_edit_str_second	36
cdd$k_edit_str_uppercase	37

(continued on next page)

Table C–7 (Cont.) Edit String Buffer Tags

Edit String Buffer Tag	Tag Value
cdd$k_edit_str_weekday_name	38
cdd$k_edit_str_year	39
cdd$k_edit_str_end	0
cdd$k_edit_str_min	cdd$k_edit_str_alphabetic
cdd$k_edit_str_max	cdd$k_edit_str_year

C.8 Expression Buffer

Describes an Oracle CDD/Repository expression. Use this buffer when the BNF for a buffer or attribute value requires an expression buffer.

Format

```
<expression>        ::=     <block_header>
                            <expression_buffer>
                            <block_terminator>

<expression_buffer> ::=   { <value_expression>         |
                            <boolean_expression>       |
                            <rse_expression>           |
                            <conditional_expression>   |
                            <table_expression>          }
                            CDD$K_EXP_END
```

Explanation of Fields

The syntax elements valid in an expression buffer are explained in the order in which they appear in the syntax diagram.

- block_header

  The header for an expression buffer uses these values for the buffer type and version numbers:

  ```
  <type>                ::= CDD$K_EXPRESSION_BUF_DSC
  <major_version_number> ::= 1
  <minor_version_number> ::= 0
  ```

- value_expression

  An expression that contains a value. The syntax of a value expression is explained in Section C.8.1.

- boolean_expression

An expression that evaluates to true or false. The syntax of a Boolean expression is explained in Section C.8.2.

- rse_expression

  An expression that specifies a record selection expression for a DSRI-compliant database. The syntax of a record selection expression is explained in Section C.8.3.

- conditional_expression

  An expression that evaluates to true or false depending on the relationship between two or more other expressions. The syntax of a conditional expression is explained in Section C.8.4.

- table_expression

  An expression that initializes values of a table or array. The syntax of a table expression is explained in Section C.8.5.

- CDD$K_EXP_END

  An unsigned word marking the end of the expression buffer.

Example

The following listing shows a metadata buffer that contains a complete expression buffer, including a buffer header and terminator.

Use this buffer as an input parameter only as a property value on **setProp** or **new**.

```
cdd$k_begin
cdd$k_metadata_buf_dsc 1 5                                    ❶
    cdd$k_entity cdd$k_ent_data_element 1 0
    cdd$k_begin
        cdd$k_directory_info_dsc 2 0                          ❷
            cdd$k_directory_name_list
                cdd$k_directory_name 7 "FIELD_1"
            cdd$k_end
            cdd$k_type cdd$k_ent_data_element
        cdd$k_eoc
        cdd$k_attribute_list
            cdd$k_attribute cdd$k_att_processing_name          ❸
            cdd$k_literal dsc$k_dtype_t 7 "FIELD_1"
            cdd$k_attribute cdd$k_att_de_datatype
            cdd$k_literal dsc$k_dtype_w 0 dsc$k_dtype_l
            cdd$k_attribute cdd$k_att_de_initial
            cdd$k_literal cdd$k_dtype_unstructured 36          ❹
```

```
            cdd$k_begin
        cdd$k_expression_buf_dsc 1 0                                    ❺
            cdd$k_exp_add                                               ❻
                cdd$k_literal dsc$k_dtype_w 0 10                        ❼
                cdd$k_exp_min                                           ❽
                cdd$k_literal dsc$k_dtype_l 0 variable1
                    cdd$k_literal dsc$k_dtype_l 0 variable2
            cdd$k_exp_end                                               ❾
            cdd$k_eoc                                                   ❿
    cdd$k_end                                                           ⓫
cdd$k_eoc                                                               ⓬
```

❶ buffer header for the metadata description buffer

❷ an embedded directory information buffer specifying the directory information for the element being defined

❸ a simple literal giving the value of the processing name attribute

❹ declaration of the unstructured literal that specifies the value of the initial value attribute

❺ buffer header for the embedded expression buffer

❻ tag beginning an arithmetic buffer requesting that the following operands be added

❼ the first operand is a simple literal 10, declared as an unsigned word

❽ tag specifying that the second operand is the smaller of two other values, each of which is specified as a longword variable

❾ tag marking the end of the arithmetic expression

❿ buffer terminator for the expression buffer

⓫ tag marking the end of the attribute list

⓬ buffer terminator for the metadata description buffer

C.8.1 Value Expressions

Value expression buffers describe value expressions you can use in Oracle CDD/Repository expression buffers.

Format

```
<value_expression>      ::=    { <arithmetic_expression>     |
                                  <dbkey_expression>         |
                                  <field_expression>         |
                                  <from_expression>          |
                                  <function_expression>      |
                                  <literal_expression>       |
                                  <statistical_expression>   |
                                  <string_expression>        |
                                  <via_expression>           |
                                  <via_table_expression>     }

<arithmetic_expression>  ::=
           { CDD$K_EXP_ADD <value_expression> <value_expression> |
             CDD$K_EXP_ASL <value_expression> <value_expression> |
             CDD$K_EXP_ASR <value_expression> <value_expression> |
             CDD$K_EXP_ONES_CMP <value_expression>               |
             CDD$K_EXP_DIV <value_expression> <value_expression> |
             CDD$K_EXP_MUL <value_expression> <value_expression> |
             CDD$K_EXP_NEG <value_expression>                    |
             CDD$K_EXP_SUB <value_expression> <value_expression> }

<dbkey_expression>  ::=  CDD$K_EXP_DBKEY <context_variable>
<field_expression>  ::=  CDD$K_EXP_ELEMENT_NAME
                        [<context_variable>] <field_name>

<context_variable>  ::=  CDD$K_EXP_CONTEXT <word_length> <mcs_string>
<field_name>        ::=  CDD$K_EXP_FIELD   <field_segment> [...]
<field_segment>     ::=  CDD$K_EXP_FIELD_SEGMENT <word_length> <mcs_string>
<from_expression>   ::=  CDD$K_EXP_FROM <value_expression> <RSE>

<function_expression>  ::=
      { CDD$K_EXP_ABS <value_expression>                       |
        CDD$K_EXP_EXP <value_expression> <value_expression>    |
        CDD$K_EXP_FAC <value_expression>                       |
        CDD$K_EXP_MOD <value_expression> <value_expression>    |
        CDD$K_EXP_RND <value_expression>                       |
        CDD$K_EXP_SGN <value_expression>                       |
        CDD$K_EXP_SQRT <value_expression>                      |
        <user_function_call>                                     }

<user_function_call> ::= CDD$K_EXP_FUNCTION
                              [<file_name>]
                               <function_name>
                               <parameter_list>
                         CDD$K_EXP_END
<file_name>       ::= CDD$K_EXP_FUNCTION_FILE <word_length> <mcs_string>
<function_name>   ::= CDD$K_EXP_FUNCTION_NAME <word_length> <mcs_string>
<parameter_list>  ::= <value_expression> [...]
```

```
<literal_expression>   ::=
         CDD$K_EXP_LITERAL
            { DSC$K_DTYPE_ADT <binary_date_time>                          |
              DSC$K_DTYPE_B   <scale> <fixed_point_value>                 |
              DSC$K_DTYPE_BU  <scale> <fixed_point_value>                 |
              DSC$K_DTYPE_W   <scale> <fixed_point_value>                 |
              DSC$K_DTYPE_WU  <scale> <fixed_point_value>                 |
              DSC$K_DTYPE_L   <scale> <fixed_point_value>                 |
              DSC$K_DTYPE_LU  <scale> <fixed_point_value>                 |
              DSC$K_DTYPE_Q   <scale> <fixed_point_value>                 |
              DSC$K_DTYPE_QU  <scale> <fixed_point_value>                 |
              DSC$K_DTYPE_O   <scale> <fixed_point_value>                 |
              DSC$K_DTYPE_OU  <scale> <fixed_point_value>                 |
              DSC$K_DTYPE_F   <floating_point_value>                      |
              DSC$K_DTYPE_D   <floating_point_value>                      |
              DSC$K_DTYPE_G   <floating_point_value>                      |
              DSC$K_DTYPE_H   <floating_point_value>                      |
              DSC$K_DTYPE_FC  <floating_point_value>                      |
              DSC$K_DTYPE_DC  <floating_point_value>                      |
              DSC$K_DTYPE_GC  <floating_point_value>                      |
              DSC$K_DTYPE_HC  <floating_point_value>                      |
              DSC$K_DTYPE_NU  <word_length> <scale> <numeric_byte_string> |
              DSC$K_DTYPE_NL  <word_length> <scale> <numeric_byte_string> |
              DSC$K_DTYPE_NLO <word_length> <scale> <numeric_byte_string> |
              DSC$K_DTYPE_NR  <word_length> <scale> <numeric_byte_string> |
              DSC$K_DTYPE_NRO <word_length> <scale> <numeric_byte_string> |
              DSC$K_DTYPE_NZ  <word_length> <scale> <numeric_byte_string> |
              DSC$K_DTYPE_P   <word_length> <scale> <numeric_byte_string> |
              DSC$K_DTYPE_T   <word_length>     <byte_string>             |
              DSC$K_DTYPE_VT  <word_length>     <byte_string>             |
              DSC$K_DTYPE_V   <longword_length> <byte_string>             |
              DSC$K_DTYPE_VU  <word_length>     <byte_string>             |
              DSC$K_DTYPE_Z   <word_length>     <byte_string>            }

<binary_date_time>       ::= quadword containing an OpenVMS standard date.
<fixed_point_value>      ::= an integer value for an attribute
<floating_point_value>   ::= a real value for an attribute
<word_length>            ::= unsigned word
<longword_length>        ::= unsigned longword
<numeric_byte_string>    ::= zero or more contiguous bytes containing
                             numeric values
<byte_string>            ::= zero or more contiguous bytes
<scale>                  ::= unsigned byte scale factor
```

```
<statistical_expression> ::=
           {CDD$K_EXP_AVG   <value_expression>  [<RSE>]      |
            CDD$K_EXP_COUNT <RSE>                           |
            CDD$K_EXP_MAX   <value_expression>  [<RSE>]      |
            CDD$K_EXP_MIN   <value_expression>  [<RSE>]      |
            CDD$K_EXP_SDV   <value_expression>   <RSE>       |
            CDD$K_EXP_TTL   <value_expression>   <RSE>       |
            CDD$K_EXP_RCT                                    |
            CDD$K_EXP_RTT   <value_expression>                 }

<string_expression>      ::=
           {CDD$K_EXP_AS2 <word_length> <mcs_string>                |
            CDD$K_EXP_ASK <word_length> <mcs_string>                |
            CDD$K_EXP_CO2 <value_expression> <value_expression>     |
            CDD$K_EXP_CO3 <value_expression> <value_expression>     |
            CDD$K_EXP_CON <value_expression> <value_expression>     |
            CDD$K_FORMAT  <value_expression> [<edit_string>]        |
            <substring_expression>                                 }

 <substring_expression>  ::=
    CDD$K_EXP_SUBSTRING <source_string> <start_position> <substring_length>

 <source_string>         ::= <value_expression>
 <start_position>        ::= <value_expression>
 <substring_length>      ::= <value_expression>
 <edit_string>           ::=
    CDD$K_EXP_EDIT_STR <word_length> <edit_string_buffer>

 <via_expression>        ::=
    CDD$K_EXP_VIA <value_expression> <RSE> <value_expression>

 <via_table_expression>  ::= CDD$K_EXP_VTB <value_expression> <table_name>
 <table_name>          ::= CDD$K_EXP_TABLE_NAME <word_length> <mcs_string>
```

Explanation of Fields

The syntax elements valid in a value expression buffer are explained in the following sections. For convenience, the syntax elements are divided by expression.

C.8.1.1 Arithmetic Expressions

The syntax elements valid in an arithmetic expression are explained in the order in which they appear in the syntax diagram.

- CDD$K_EXP_ADD

  An unsigned word indicating that Oracle CDD/Repository adds the first value expression to the second value expression.

- CDD$K_EXP_ASL

An unsigned word indicating that Oracle CDD/Repository performs an arithmetic shift left. The operation shifts the bits in the first value expression to the right by the amount specified by the second value expression.

- CDD$K_EXP_ASR

  An unsigned word indicating that Oracle CDD/Repository performs an arithmetic shift right. The operation shifts the bits in the first value expression to the right by the amount specified by the second value expression.

- CDD$K_EXP_ONES_CMP

  An unsigned word indicating that Oracle CDD/Repository performs the one's complement operation on a value expression.

- CDD$K_EXP_DIV

  An unsigned word indicating that Oracle CDD/Repository divides the first value expression by the second value expression.

- CDD$K_EXP_MUL

  An unsigned word indicating that Oracle CDD/Repository multiplies the first value expression by the second value expression.

- CDD$K_EXP_NEG

  An unsigned word indicating that Oracle CDD/Repository provides the negative of the value expression.

- CDD$K_EXP_SUB

  An unsigned word indicating that Oracle CDD/Repository subtracts the second value expression from the first value expression.

C.8.1.2 Database Key Expressions and Field Expressions

A field expression specifies the name of a field in the repository. The name of the field consists of either a context variable and field name or the field name only.

A database key (dbkey) expression retrieves a specific record from an Oracle Rdb database by using an internal system pointer. Programs that call Oracle CDD/Repository can locate database records by evaluating the dbkey expression.

The syntax elements valid in a dbkey expression are explained in the order in which they appear in the syntax diagram.

- CDD$K_EXP_DBKEY

  an unsigned word that indicates the beginning of a dbkey expression

- context_variable

  a variable that identifies the desired record instance

  Each database key identifies only one record, but a record might have more than one database key.

- CDD$K_EXP_ELEMENT_NAME

  an unsigned word marking the beginning of a field expression

- field_expression

  an expression that provides the value of a particular field in a record instance

  Wherever a field expression is valid, a dbkey expression is valid.

C.8.1.3 From Expressions

The syntax elements valid in a from expression are explained in the order in which they appear in the syntax diagram.

- CDD$K_EXP_FROM

  An unsigned word indicating that a from_expression follows.

- value_expression

  An expression to be evaluated at run time using values from the record selection expression.

- RSE

  A record selection expression identifying the records to be used at run time to calculate the value of the value_expression. The syntax of a record selection expression is explained in Section C.8.3.

You can describe an expression comprising both a record selection expression and a value expression. At run time, Oracle CDD/Repository evaluates such an expression with the following steps:

1. Uses the record selection expression to establish a record stream.
2. Evaluates the value expression:
   - If the record stream is not empty, Oracle CDD/Repository uses the field values from the first record in the stream to evaluate the expression and uses the result of this evaluation as the value of the from expression.

– If the record stream is empty, the from expression has no value; it produces an error.

C.8.1.4 Function Expressions

Oracle CDD/Repository expression buffers let you express some commonly used functions. To express less common functions, you can use the special function tag, CDD$K_EXP_FUNCTION.

The syntax elements valid in a function expression are explained in the order in which they appear in the syntax diagram.

- CDD$K_EXP_ABS

  An unsigned word requesting the absolute value of a value expression.

- CDD$K_EXP_EXP

  An unsigned word requesting that Oracle CDD/Repository raise the first value expression to the power specified by the second value expression.

- CDD$K_EXP_FAC

  An unsigned word requesting that Oracle CDD/Repository perform the factorial operation on the value expression.

- CDD$K_EXP_MOD

  An unsigned word requesting that Oracle CDD/Repository perform the modulus (MOD) operation. Oracle CDD/Repository divides the first value expression by the second, returning the remainder.

- CDD$K_EXP_RND

  An unsigned word requesting that Oracle CDD/Repository round a value expression to the nearest whole number.

- CDD$K_EXP_SGN

  An unsigned word requesting that Oracle CDD/Repository test the sign of the value expression. CDD$K_EXP_SGN returns –1 if the value expression is less than zero, returns 0 if the value expression equals zero, and returns 1 if the value expression exceeds zero.

- CDD$K_EXP_SQRT

  An unsigned word requesting that Oracle CDD/Repository return the square root of a value expression.

- user_function_call

  An expression that allows your program to specify a function Oracle CDD/Repository does not provide.

- CDD$K_EXP_FUNCTION

  An unsigned word marking the beginning of a user-defined function.

- CDD$K_EXP_FUNCTION_FILE

  An unsigned word marking the beginning of the name of the file containing the function you want to invoke.

- word_length

  An unsigned word giving the length in bytes of the file name or function name that follows.

- mcs_string

  A string of one or more contiguous bytes giving the name of the file that contains the function you want to invoke or the name of the function itself.

- CDD$K_EXP_FUNCTION_NAME

  An unsigned word marking the beginning of the name of the function you want to invoke.

  For example, to express the string function that converts to uppercase, use the CDD$K_EXP_FUNCTION with the following arguments:

```
CDD$K_EXP_FUNCTION
    CDD$K_EXP_FUNCTION_NAME, 9, 'FN$UPCASE'
    CDD$K_EXP_ELEMENT_NAME,     1, 'A'
    CDD$K_EXP_ELEMENT_NAME,     1, 'B'
CDD$K_EXP_END
```

C.8.1.5 Literal Expressions

A literal expression is a simple data value. You can describe simple data values of any legal Oracle CDD/Repository data type.

The syntax elements valid in a literal expression are explained in the order in which they appear in the syntax diagram.

- CDD$K_EXP_LITERAL

  An unsigned word marking the beginning of the literal.

- numeric_byte_string

  Bytes containing the value of the attribute in numeric format; there must be exactly as many bytes as indicated in the preceding length field.

- byte_string

  Bytes containing the value of the attribute; there must be exactly as many bytes as indicated in the preceding length field.

C.8.1.6 Statistical Expressions

Depending on a language syntax, these statistical operators may operate on simple value expressions, or on a value expression for a record in a record stream (if used with an RSE).

For example, CDD$K_EXP_AVG returns the average value of the specified value expressions. If used with an RSE, it averages values for each record in the record stream.

The syntax elements valid in a statistical expression are explained in the order in which they appear in the syntax diagram.

- CDD$K_EXP_AVG

  An unsigned word requesting that Oracle CDD/Repository return the average value of the specified value expressions or of all records in the record stream.

- CDD$K_EXP_COUNT

  An unsigned word requesting that Oracle CDD/Repository return the count of the number of items across records in a record stream.

- CDD$K_EXP_MAX

  An unsigned word requesting that Oracle CDD/Repository return the larger value of two value expressions, or across a record stream.

- CDD$K_EXP_MIN

  An unsigned word requesting that Oracle CDD/Repository return the smaller of two value expressions, or the smallest value in a record stream.

- CDD$K_EXP_SDV

  An unsigned word requesting that Oracle CDD/Repository compute the standard deviation of the values in the specified record stream.

- CDD$K_EXP_TTL

  An unsigned word requesting that Oracle CDD/Repository return the sum of all values in the given expressions for each record in the record stream.

- CDD$K_EXP_RCT

  An unsigned word requesting that Oracle CDD/Repository keep a running count of the number of times it has evaluated the specified item for the DATATRIEVE PRINT statement.

- CDD$K_EXP_RTT

  An unsigned word requesting that Oracle CDD/Repository keep a running total of the value statement for each time it has evaluated the DATATRIEVE PRINT statement.

C.8.1.7 String Expressions

A string expression performs an operation on a character string or on a value expression that can be interpreted as a character string.

The syntax elements valid in a string expression are explained in the order in which they appear in the syntax diagram.

- CDD$K_EXP_AS2

  An unsigned word specifying a double star prompt for DATATRIEVE. At run time, prompts the user for the value of the *field* variable once.

- CDD$K_EXP_ASK

  An unsigned word specifying a single star prompt for DATATRIEVE. At run time, DATATRIEVE uses this string to prompt the user for the value of the field expression each time the field variable is used to calculate another value.

- CDD$K_EXP_CO2

  An unsigned word requesting that Oracle CDD/Repository convert the values of each value expression to a character string literal and append the first string onto the second. Oracle CDD/Repository suppresses trailing spaces of the first string and does nothing to the second string.

- CDD$K_EXP_CO3

  An unsigned word requesting that Oracle CDD/Repository perform a triple concatenation operation. Oracle CDD/Repository converts the values of each value expression to character string literals. Oracle CDD/Repository appends the second string to the first, suppressing trailing spaces of the first value expression. It then inserts a space between the two expressions and does nothing to the second value expression.

- CDD$K_EXP_CON

  An unsigned word requesting that Oracle CDD/Repository perform a simple concatenation. Oracle CDD/Repository converts the values of each value expression to character string literals, then appends the second value expression to the first. It does not remove spaces from either string.

- CDD$K_EXP_EDIT_STR

  An unsigned word indicating that an edit string for the field follows. You describe an edit string with an edit string buffer. Refer to Section C.7 for more information.

- CDD$K_EXP_SUBSTRING

  An unsigned word requesting that Oracle CDD/Repository extract a substring from a source string.

- source_string

  A value expression specifying the string from which the substring is to be extracted. It is treated as a character string literal.

- start_position

  A value expression specifying the character position in the source string at which substring extraction begins.

- substring-length

  A value expression specifying the number of characters to extract from the source string.

C.8.1.8 VIA Expressions

A VIA expression is similar to a from expression, except that a VIA expression provides a default value if the record stream is empty.

The syntax elements valid in a VIA expression are explained in the order in which they appear in the syntax diagram.

- CDD$K_EXP_VIA

  An unsigned word marking the beginning of the via_expression.

- value_expression

  The first value expression identifies the value you want to calculate using values from the record stream. The second value expression specifies a default value to return if the record stream is empty.

- RSE

  Specifies the records you want to use to evaluate the first value expression.

At run time, the database system establishes a record stream by using the record selection expression.

- If the record stream is not empty, the database system uses values from the first record in the stream to evaluate the expression. The result of this evaluation is the value of the VIA expression.
- If the record stream is empty, the database system uses the second value expression as the value of the VIA expression.

C.8.1.9 VIA Table Expressions

The VIA table expression describes an expression that looks up an entry in a DATATRIEVE table.

The syntax elements valid in a table expression are explained in the order in which they appear in the syntax diagram.

- CDD$K_EXP_VTB

  an unsigned word marking the beginning of the table expression

- value_expression

  the expression describing the entry you want to look up

- CDD$K_EXP_TABLE_NAME

  an unsigned word marking the beginning of the table name

- word_length

  an unsigned word giving the length in bytes of the table name that follows

- mcs_string

  one or more contiguous bytes giving the name of the table in which you want to look up the value expression

C.8.2 Boolean Expressions

Boolean expression buffers describe the Boolean expressions you can insert into an Oracle CDD/Repository expression buffer. A Boolean expression can be a relational expression, a literal expression, a logical expression, or a record selection expression.

Format

```
<boolean_expression>       ::= { <relational_expression>   |
                                 <literal_expression>      |
                                 <logical_expression>      |
                                 <rse_expression>           }
```

Explanation of Fields

The syntax elements valid in a Boolean expression are explained in the order in which they appear in the syntax diagram.

- relational_expression

  An expression that returns true or false based on the relation between one or more value expressions. The syntax of a relational expression is explained in Section C.8.2.1.

- literal_expression

  A literal expression. The syntax of a literal expression is explained in Section C.8.1.5.

- logical_expression

  An expression that returns true or false based on a specified logical condition. The syntax of a logical expression is explained in Section C.8.2.2.

- rse_expression

  An expression that evaluates a record stream to determine whether any records or one, and only one, record are present. The syntax of the rse_expression is explained in Section C.8.3.

C.8.2.1 Relational Expressions

Relational expression buffers describe the format of the Oracle CDD/Repository relational expressions.

Format

```
<relational_expression> ::=
   {  CDD$K_EXP_BET <value_expression> <value_expression> <value_expression> |
      CDD$K_EXP_COT <value_expression> <value_expression>      |
      CDD$K_EXP_EQL <value_expression> <value_expression>      |
      CDD$K_EXP_GTR <value_expression> <value_expression>      |
      CDD$K_EXP_GEQ <value_expression> <value_expression>      |
      CDD$K_EXP_INT <value_expression> <table_name>            |
      CDD$K_EXP_LSS <value_expression> <value_expression>      |
      CDD$K_EXP_LEQ <value_expression> <value_expression>      |
      CDD$K_EXP_MATCHES <value_expression> <value_expression>  |
      CDD$K_EXP_NEQ <value_expression> <value_expression>      |
      CDD$K_EXP_STW <value_expression> <value_expression>       }
```

Explanation of Fields

The syntax elements valid in a relational expression are explained in the order in which they appear in the syntax diagram.

- CDD$K_EXP_COT

  An unsigned word requesting the containing operation. CDD$K_EXP_ COT returns true if the first value expression contains the second value expression as a substring. Oracle CDD/Repository interprets both value expressions as strings. String comparisons are not case sensitive.

- CDD$K_EXP_EQL

  An unsigned word requesting that Oracle CDD/Repository return true if the first value expression has the same value as the second value expression. String comparisons are case sensitive.

- CDD$K_EXP_GEQ

  An unsigned word requesting that Oracle CDD/Repository return true if the value of the first value expression is greater than or equal to the value of the second. String comparisons are case sensitive.

- CDD$K_EXP_GTR

  An unsigned word requesting that Oracle CDD/Repository return true if the value of the first value expression is greater than the value of the second. String comparisons are case sensitive.

- CDD$K_EXP_INT

  An unsigned word requesting that Oracle CDD/Repository return true if the first field name is in the table specified by the second field name. Table names are explained in Section C.8.5.

- CDD$K_EXP_LEQ

  An unsigned word requesting that Oracle CDD/Repository return true if the value of the first value expression is less than or equal to the value of the second value expression. String comparisons are case sensitive.

- CDD$K_EXP_LSS

  An unsigned word requesting that Oracle CDD/Repository return true if the value of the first value expression is less than the value of the second. String comparisons are case sensitive.

- CDD$K_EXP_NEQ

  An unsigned word requesting that Oracle CDD/Repository return true if the value of the first value expression is not equal to the value of the second value expression. String comparisons are case sensitive.

- CDD$K_EXP_STW

  An unsigned word requesting that Oracle CDD/Repository return true if the first string starts with the second. Operands must be string expressions. Comparisons are case sensitive.

Description

The value expressions you supply should have values of compatible data types. Oracle CDD/Repository performs the necessary conversions.

C.8.2.2 Logical Expressions

Logical expression buffers describe the format of the logical expressions you can use in an expression buffer.

Format

```
<logical_expression>   ::=
      {CDD$K_EXP_AND <boolean_expression> <boolean_expression>   |
       CDD$K_EXP_OR  <boolean_expression> <boolean_expression>   |
       CDD$K_EXP_MIS <value_expression>                          |
       CDD$K_EXP_NOT <boolean_expression>                        }
```

Explanation of Fields

The syntax elements valid in logical expression are explained in the order in which they appear in the syntax diagram.

- CDD$K_EXP_AND

  An unsigned word requesting that Oracle CDD/Repository perform the logical AND of two Boolean expressions.

- CDD$K_EXP_OR

  An unsigned word requesting that Oracle CDD/Repository perform the logical OR of two Boolean expressions.

- CDD$K_EXP_MIS

  An unsigned word requesting that Oracle CDD/Repository test the value of a field to see if it matches the missing value. CDD$K_EXP_MIS returns true if the value for the field expression is the same as the value of the CDD$DATA_ELEMENT_MISSING_VALUE attribute for the field. If the field value differs from the missing value, or if the field does not have a missing value, this function returns false.

- CDD$K_EXP_NOT

  An unsigned word requesting that Oracle CDD/Repository perform the logical NOT of a Boolean expression.

C.8.3 Record Selection Expressions

Record selection expression (RSE) buffers describe the record selection expressions you can insert into an Oracle CDD/Repository expression buffer. A record selection expression defines a stream of records.

The RSE refers to one or more Oracle Rdb relations and explicitly declares a context variable for each, thereby producing a stream of records that meet the selection criteria you supply in the remainder of the RSE.

Format

```
<rse_expression> ::=  CDD$K_EXP_ANY <RSE> | CDD$K_EXP_UNQ <RSE>

<RSE>            ::=     CDD$K_EXP_RSE
                       { [ <context_variable>] <RSE_source> }
                         [ <rse_clause> ]
                           CDD$K_EXP_END

<RSE_source>     ::=
      { CDD$K_EXP_RELATION    <word_length> <mcs_string>  |
        CDD$K_EXP_RELATION_ID <relation_id>               |
        CDD$K_EXP_DOMAIN      <word_length> <mcs_string>  |
        CDD$K_EXP_COLLECTION  <word_length> <mcs_string>  |
        CDD$K_EXP_LIST        <word_length> <mcs_string>  |
        CDD$K_EXP_RECORD      <word_length> <mcs_string>    }

<rse_clause>     ::= [: <cross>                           |
                        <all_or_first>                    |
                        <with>                            |
                        <reduce>                          |
                        <sort>                              :]
```

```
<cross>          ::=     CDD$K_EXP_CROSS
                       { [ <context_variable> ] <RSE_source>
                         [ CDD$K_EXP_OVER       <field_expression> ] } ...
                           CDD$K_EXP_END

<all_or_first>   ::= { CDD$K_EXP_ALL | CDD$K_EXP_FIRST <value_expression> }

<with>           ::= CDD$K_EXP_BOOLEAN <boolean_expression>

<reduce>         ::= CDD$K_EXP_REDUCE
                       <value_expression> [...]
                     CDD$K_EXP_END

<sort>           ::=     CDD$K_EXP_SORT
                       { [ CDD$K_EXP_ASCENDING | CDD$K_EXP_DESCENDING ]
                           <value_expression> } [...]
                           CDD$K_EXP_END

<context_variable>  ::= CDD$K_EXP_CONTEXT <word_length> <mcs_string>
<relation_id>       ::= unsigned byte
<mcs_string>        ::= byte string
<word_length>       ::= unsigned byte
```

Explanation of Fields

The syntax elements valid in a record selection expression are explained in the order in which they appear in the syntax diagram.

- CDD$K_EXP_ANY

  An unsigned word requesting that if there are any records in the specified record stream, Oracle CDD/Repository return a value of true.

- CDD$K_EXP_UNQ

  An unsigned word requesting that if there is one and only one record in the specified record stream, Oracle CDD/Repository return a value of true.

- CDD$K_EXP_RSE

  An unsigned word marking the beginning of the record selection expression.

- context_variable

  A temporary name associated with the named relation. The context variable identifies the desired record instance.

- CDD$K_EXP_CONTEXT

  An unsigned word indicating that a context variable follows.

- RSE_source

  A clause indicating where the RSE comes from.

- CDD$K_EXP_RELATION

  An unsigned word indicating the relation in which the record selection expression is defined. The name of the relation follows.

- CDD$K_EXP_RELATION_ID

  An unsigned word indicating the relation in which the record selection expression is defined; the relation is identified by an unsigned byte value representing the relation.

- CDD$K_EXP_DOMAIN

  An unsigned word indicating the DATATRIEVE domain in which the record selection expression is defined. The name of the domain follows.

- CDD$K_EXP_COLLECTION

  An unsigned word indicating the DATATRIEVE collection in which the record selection expression is defined. The name of the collection follows.

- CDD$K_EXP_LIST

  An unsigned word indicating the list in which the record selection expression is defined. The name of the list follows.

- CDD$K_EXP_RECORD

  An unsigned word indicating the record that defines the record selection expression. The name of the record follows.

- rse_clause

  Specifies the selection criteria for records from the named relations in a DSRI-compliant database.

- CDD$K_EXP_CROSS

  An unsigned word requesting that the relations that follow be combined in a relational join operation.

- CDD$K_EXP_OVER

  An unsigned word indicating that the relations are to be joined using the values in a particular field.

- field_expression

  An expression identifying the field that is to be used to combine the relations.

- CDD$K_EXP_ALL

  An unsigned word requesting all records that meet the selection criteria.

- CDD$K_EXP_FIRST

  An unsigned word requesting at most the number of records specified by the value expression. If fewer records than the specified value meet the selection criteria, the database software returns the entire record stream.

  If the value expression is less than or equal to zero, the record stream is empty.

- CDD$K_EXP_BOOLEAN

  An unsigned word requesting that the record stream include only those records that result in an evaluation of true for the Boolean expression.

- CDD$K_EXP_REDUCE

  An unsigned word requesting that the records in the record stream include only one instance of each unique field value for the specified field or fields. The field whose value is to be evaluated is called the reduce key. The value expression can be any field name or related value expression; it must be unique, but it can be the result of a calculation based on other value expressions.

  Two values are equal if they evaluate to the same value or they both evaluate to missing. The reduction_count parameter specifies the number of value expressions to reduce to.

- CDD$K_EXP_SORT

  Sorts the retrieved records on the basis of the field identified by the value expression.

  The item_count parameter specifies the number of sort items that are being sorted.

- CDD$K_EXP_ASCENDING

  An unsigned word indicating that the records are to be sorted in ascending order, that is, with lowest values first and highest values last.

- CDD$K_EXP_DESCENDING

  An unsigned word indicating that the records are to be sorted in descending order, that is, with highest values first and lowest values last.

- mcs_string

  One or more contiguous bytes containing printable characters from the Digital Multinational character set.

- word_length

  An unsigned word indicating the length of the character string that follows.

- CDD$K_EXP_END

  An unsigned word indicating the end of the CROSS clause.

Description

If you supply more than one relation in the RSE, Oracle CDD/Repository performs a relational join of those relations before evaluating the remainder of the RSE.

The order in which the RSE clauses appear in the request buffer is unimportant. DSRI-compliant database software always evaluates the clauses in the following order:

- CDD$K_EXP_ALL or CDD$K_EXP_FIRST
- CDD$K_EXP_BOOLEAN
- CDD$K_EXP_REDUCE
- CDD$K_EXP_SORT

Unless a sort clause is present, the order in which records are fetched is undefined.

The following rules apply to sorting:

- For the first sort_item, the default sort order is ASCENDING.
- For the second or subsequent sort_items, if you supply neither ASCENDING nor DESCENDING, Oracle CDD/Repository uses the sort order for the preceding sort_item.
- Missing values are always ordered as though they had a greater value than every other value.

C.8.4 Conditional Expressions

Conditional expression buffers describe the conditional expressions you can insert into an Oracle CDD/Repository expression buffer. A conditional expression returns a value based on the value of a Boolean expression.

Format

```
<conditional_expression> ::= CDD$K_EXP_COND
                               <if_clause>
                               <then_clause>
                             [ <else_clause> ]

<if_clause>          ::= CDD$K_EXP_IF   <boolean_expression>
<then_clause>        ::= CDD$K_EXP_THEN <value_expression>
<else_clause>        ::= CDD$K_EXP_ELSE <value_expression>
```

Explanation of Fields

The syntax elements valid in conditional expressions are explained in the order in which they appear in the syntax diagram.

- CDD$K_EXP_COND

  an unsigned word marking the beginning of the conditional expression

- CDD$K_EXP_IF

  an unsigned word marking the beginning of the condition to be tested

- boolean_expression

  an expression identifying the condition to be tested

- CDD$K_EXP_THEN

  an unsigned word tag indicating that a value expression follows

- value_expression

  a value expression to be evaluated if the condition specified by the if_clause is true

- CDD$K_EXP_ELSE

  an unsigned word tag indicating the presence of an optional second value expression to be evaluated if the condition specified by the if_clause evaluates to false

Description

Conditional expressions may be nested.

Example

```
discount_price computed by
    choice
        price lt 20000 then (price * .9)
        price lt 30000 then (price * .8)
        price lt 40000 then (price * .7)
        else price * .6
    end_choice
```

The following DATATRIEVE field definition illustrates the use of conditional expressions.

```
CDD$K_BEGIN
CDD$K_EXPRESSION_BUF_DSC 1 0
       CDD$K_EXP_COND
           CDD$K_EXP_IF
               CDD$K_EXP_LSS
                   CDD$K_EXP_ELEMENT_NAME
                       CDD$K_EXP_FIELD
                           CDD$K_EXP_FIELD_SEGMENT 5 price
                   DSC$K_DTYPE_LU 0 20000

           CDD$K_EXP_THEN
               CDD$K_EXP_MUL
                   CDD$K_EXP_ELEMENT_NAME
                       CDD$K_EXP_FIELD
                           CDD$K_EXP_FIELD_SEGMENT 5 price
                   DSC$K_DTYPE_LU, 0, .9

           CDD$K_EXP_ELSE
               CDD$K_EXP_COND
                   CDD$K_EXP_IF
                       CDD$K_EXP_LSS
                           CDD$K_EXP_ELEMENT_NAME
                               CDD$K_EXP_FIELD
                                   CDD$K_EXP_FIELD_SEGMENT 5 price
                           DSC$K_DTYPE_LU, 0, 30000

                   CDD$K_EXP_THEN
                       CDD$K_EXP_MUL,
                           CDD$K_EXP_ELEMENT_NAME
                               CDD$K_EXP_FIELD
                                   CDD$K_EXP_FIELD_SEGMENT 5 price
                           DSC$K_DTYPE_LU, 0, .8

                   CDD$K_EXP_ELSE
                       CDD$K_EXP_COND
                           CDD$K_EXP_IF
                               CDD$K_EXP_LSS
                                   CDD$K_EXP_ELEMENT_NAME
                                       CDD$K_EXP_FIELD
                                           CDD$K_EXP_FIELD_SEGMENT 5 price
                                   DSC$K_DTYPE_LU, 0, 40000

                           CDD$K_EXP_THEN
                               CDD$K_EXP_MUL
                                   CDD$K_EXP_ELEMENT_NAME
                                       CDD$K_EXP_FIELD
                                           CDD$K_EXP_FIELD_SEGMENT 5 price
                                   DSC$K_DTYPE_LU, 0, .7
```

```
                        CDD$K_EXP_ELSE
                            CDD$K_EXP_MUL
                                CDD$K_EXP_ELEMENT_NAME
                                    CDD$K_EXP_FIELD
                                        CDD$K_EXP_FIELD_SEGMENT 5 price
                                DSC$K_DTYPE_LU, 0, .6
                    CDD$K_EXP_END
            CDD$K_EXP_END
    CDD$K_EXP_END
CDD$K_EOC
```

To describe the DATATRIEVE field in Oracle CDD/Repository, create a data element called DISCOUNT_PRICE owning the CDD$DATA_ELEMENT_COMPUTED_VALUE relationship. The relationship points to a CDD$DATA_VALUE entity, which owns a CDD$DATA_VALUE_EXPRESSION attribute describing the conditional expression. This example shows the expression buffer for it.

C.8.5 Table Expressions

Table expressions declare individual initial or missing values for each element in an array.

Format

```
<table>    ::=    CDD$K_EXP_TABLE
                { <value_expression> | <boolean_expression> } ...
                  CDD$K_EXP_END
```

Explanation of Fields

The syntax elements valid in a table expression are explained in the order in which they appear in the syntax diagram.

- CDD$K_EXP_TABLE

  an unsigned word marking the beginning of a table expression

- value_expression

  a list of one or more expressions giving values for elements in an array

- boolean_expression

  a list of one or more Boolean expressions giving values of true or false for the elements of an array

- CDD$K_EXP_END

  an unsigned word tag marking the end of the table expression

Description

A single expression buffer can describe a list of expressions in a tabular format.

C.8.6 Expression Buffer Tags

Expression buffer tags are unsigned words. Table C–8 shows the value of each expression buffer tag.

Table C–8 Expression Buffer Tags

Expression Buffer Tag	Tag Value
cdd$k_exp_abs	1
cdd$k_exp_add	2
cdd$k_exp_all	3
cdd$k_exp_and	4
cdd$k_exp_any	5
cdd$k_exp_ascending	6
cdd$k_exp_ask	7
cdd$k_exp_asl	8
cdd$k_exp_asr	9
cdd$k_exp_as2	10
cdd$k_exp_avg	11
cdd$k_exp_boolean	12
cdd$k_exp_collection	13
cdd$k_exp_con	14
cdd$k_exp_cond	15
cdd$k_exp_context	16
cdd$k_exp_cot	17
cdd$k_exp_count	18
cdd$k_exp_co2	19
cdd$k_exp_co3	20
cdd$k_exp_cross	76
cdd$k_exp_dbkey	21
cdd$k_exp_descending	22
cdd$k_exp_div	23
cdd$k_exp_domain	24

(continued on next page)

Table C–8 (Cont.) Expression Buffer Tags

Expression Buffer Tag	Tag Value
cdd$k_exp_edit_str	77
cdd$k_exp_element_name	25
cdd$k_exp_else	26
cdd$k_exp_end	0
cdd$k_exp_eql	27
cdd$k_exp_exp	28
cdd$k_exp_fac	29
cdd$k_exp_field	30
cdd$k_exp_field_segment	75
cdd$k_exp_first	31
cdd$k_exp_format	78
cdd$k_exp_from	32
cdd$k_exp_function	33
cdd$k_exp_function_file	34
cdd$k_exp_function_name	35
cdd$k_exp_geq	36
cdd$k_exp_gtr	37
cdd$k_exp_if	38
cdd$k_exp_int	39
cdd$k_exp_leq	40
cdd$k_exp_list	41
cdd$k_exp_literal	42
cdd$k_exp_lss	43
cdd$k_exp_matches	44
cdd$k_exp_max	45
cdd$k_exp_min	46
cdd$k_exp_mis	47
cdd$k_exp_mod	48
cdd$k_exp_mul	49

(continued on next page)

Table C–8 (Cont.) Expression Buffer Tags

Expression Buffer Tag	Tag Value
cdd$k_exp_neg	50
cdd$k_exp_neq	51
cdd$k_exp_not	52
cdd$k_exp_ones_cmp	53
cdd$k_exp_or	54
cdd$k_exp_rct	55
cdd$k_exp_record	56
cdd$k_exp_reduce	57
cdd$k_exp_relation	58
cdd$k_exp_relation_id	59
cdd$k_exp_rse	60
cdd$k_exp_rnd	61
cdd$k_exp_rtt	62
cdd$k_exp_sdv	63
cdd$k_exp_sgn	64
cdd$k_exp_sort	65
cdd$k_exp_sqrt	66
cdd$k_exp_stw	67
cdd$k_exp_sub	68
cdd$k_exp_table	69
cdd$k_exp_then	70
cdd$k_exp_ttl	71
cdd$k_exp_unq	72
cdd$k_exp_via	73
cdd$k_exp_vtb	81
cdd$k_exp_xor	74
cdd$k_exp_over	79

D

Protocol Validations

This appendix summarizes the rules Oracle CDD/Repository enforces when you store record and field definitions in Oracle CDD/Repository.

If you use the Oracle CDD/Repository callable interface to create a definition that violates one of these rules, Oracle CDD/Repository returns an error. The error includes the name of the validation rule you violated.

Each section in the appendix describes one validation rule. The validation rules appear in alphabetical order.

D.1 CDD$AGG_ALIGN_VAL

The value of the CDD$DATA_AGGREGATE_ALIGNMENT attribute must be a positive integer.

D.2 CDD$ARRAY_ORDER_VAL

The value of the CDD$DATA_ARRAY_ORDER attribute must be a positive integer.

D.3 CDD$DATA_DIM_HIGH

A CDD$DATA_DIMENSION entity must own an instance of either the CDD$DATA_DIMENSION_HIGH_BOUND attribute or the CDD$DATA_DIMENSION_HIGH_BD_REL relationship, but not both.

D.4 CDD$DATA_DIM_LOW

No CDD$DATA_DIMENSION entity can own instances of both the CDD$DATA_DIMENSION_LOW_BOUND attribute and the CDD$DATA_DIMENSION_LOW_BD_REL relationship.

D.5 CDD$DATATYPE_VAL

The value of the CDD$DATA_ELEMENT_DATATYPE attribute must be an OpenVMS standard data type or one of the following values:

CDD$K_DTYPE_ALPHABETIC
CDD$K_DTYPE_POINTER
CDD$K_DTYPE_SEG_STRING

D.6 CDD$DEPEND_ID_VAL

The value of the CDD$DATA_VALUE_DEPENDENCY_ID attribute must be a positive integer.

D.7 CDD$DIGITS_LENGTH

If a data element owns both the CDD$DATA_ELEMENT_LENGTH and CDD$DATA_ELEMENT_DIGITS attribute, their values must conform to the following rules:

CDD$DATA_ELEMENT_DATATYPE	Rule
DSC$K_DTYPE_NZ DSC$K_DTYPE_NLO DSC$K_DTYPE_NRO DSC$K_DTYPE_NU	length = digits
DSC$K_DTYPE_NL DSC$K_DTYPE_NR	length = digits—1
DSC$K_DTYPE_P	length = digits/2 + 1 (with truncation)

In addition to the above restrictions, the data type constrains the value of the CDD$DATA_ELEMENT_DIGITS attribute, as follows:

CDD$DATA_ELEMENT_DATATYPE	Range of Values for CDD$DATA_ELEMENT_DIGITS
DSC$K_DTYPE_B	1—3
DSC$K_DTYPE_BU	1—4
DSC$K_DTYPE_W	1—5
DSC$K_DTYPE_WU	1—6
DSC$K_DTYPE_L	1—10

CDD$DATA_ELEMENT_DATATYPE	Range of Values for CDD$DATA_ELEMENT_DIGITS
DSC$K_DTYPE_LU	1—10
DSC$K_DTYPE_Q	1—20
DSC$K_DTYPE_QU	1—19
DSC$K_DTYPE_O	1—39
DSC$K_DTYPE_OU	1—39

D.8 CDD$DTYPE_DIG_SCALE

The CDD$DATA_ELEMENT_SCALE and the CDD$DATA_ELEMENT_DIGITS attributes may exist only for data elements whose CDD$DATA_ELEMENT_ DATATYPE attribute is numeric. CDD$DATA_ELEMENT_SCALE and CDD$DATA_ELEMENT_DIGITS *cannot* exist for data elements with the following data types:

DSC$K_DTYPE_ADT
DSC$K_DTYPE_T
DSC$K_DTYPE_V
DSC$K_DTYPE_VT
DSC$K_DTYPE_VU
DSC$K_DTYPE_Z
CDD$K_DTYPE_ALPHABETIC
CDD$K_DTYPE_POINTER
CDD$K_DTYPE_SEG_STRING

D.9 CDD$DTYPE_JUSTIFY

The CDD$DATA_ELEMENT_JUSTIFICATION attribute may exist only for entities whose CDD$DATA_ELEMENT_DATATYPE attribute is text (DSC$K_DTYPE_T) or alphabetic (CDD$K_DTYPE_ALPHABETIC).

D.10 CDD$DTYPE_LENGTH

If a data element owns the CDD$DATA_ELEMENT_DATATYPE attribute, and the value of the attribute is one of the following, the data element must also own the CDD$DATA_ELEMENT_LENGTH attribute:

DSC$K_DTYPE_T
DSC$K_DTYPE_V
DSC$K_DTYPE_VT
DSC$K_DTYPE_VU

DSC$K_DTYPE_Z
CDD$K_DTYPE_ALPHABETIC

D.11 CDD$DV_ALL_NONE

Within a CDD$DATA_OVERLAY_AGGREGATE entity, if any CDD$DATA_OVERLAY entity owns the CDD$DATA_OVERLAY_IDENTIFICATION relationship, all CDD$DATA_OVERLAY entities must own it.

D.12 CDD$INPUT_PROMPT_VAL

The value of the CDD$DATA_ELEMENT_INPUT_PROMPT attribute must contain only bytes from the Digital Multinational character set.

D.13 CDD$INST_PATH

If a data instance entity owns an instance of the CDD$DATA_INSTANCE_PATH relationship, it must also own an instance of the CDD$DATA_INSTANCE_ROOT relationship.

D.14 CDD$JUSTIFY_VAL

The value of the CDD$DATA_ELEMENT_JUSTIFICATION attribute must be one of the following:

CDD$K_JUSTIFIED_RIGHT
CDD$K_JUSTIFIED_LEFT
CDD$K_JUSTIFIED_CENTER

D.15 CDD$ONE_INST_ROOT

Each CDD$DATA_INSTANCE entity can own no more than one instance of the CDD$DATA_INSTANCE_ROOT relationship.

D.16 CDD$OUT_HEAD_VAL

In a buffer you pass to the callable interface, the value of the CDD$DATA_ELEMENT_OUTPUT_HEADER attribute must be expressed as a text buffer.

D.17 CDD$PATH_STEP_VAL

The value of the CDD$DATA_INSTANCE_PATH_STEP attribute must be a positive integer.

D.18 CDD$REQ_DATA_VAL_EXP

If a data value entity does not own the CDD$DATA_VALUE_EXPRESSION attribute, it must own the CDD$DATA_VALUE_DEPENDS_ON relationship.

D.19 CDD$REQ_INIT_VALUE

A data element cannot own both the CDD$DATA_ELEMENT_INITIAL_ VALUE attribute and the CDD$DATA_ELEMENT_INITIAL_DEF relationship.

D.20 CDD$REQ_MISS_VALUE

A data element cannot own both the CDD$DATA_ELEMENT_MISSING_ VALUE attribute and the CDD$DATA_ELEMENT_MISSING_DEF relationship.

D.21 CDD$REQ_PTR_REF

The CDD$DATA_ELEMENT_POINTER_REF relationship may exist only for data elements whose CDD$DATA_ELEMENT_DATATYPE attribute is pointer (CDD$K_DTYPE_POINTER).

D.22 CDD$REQ_SEG_STRING

If a data element owns an instance of the CDD$DATA_ELEMENT_SEG_ SUBTYPE or CDD$DATA_ELEMENT_SEGMENT_LENGTH attributes, then it must also own the CDD$DATA_ELEMENT_DATATYPE attribute with a value of CDD$K_DTYPE_SEG_STRING.

D.23 CDD$SEQ_NUM_VAL

The value of the CDD$DATA_SEQUENCE_NUMBER attribute must be a positive integer.

D.24 CDD$UNIQ_ARRAY_ORDER

If a data element or data aggregate owns more than one instance of the CDD$DATA_ARRAY_HAS_DIMENSION relationship, the value of the CDD$DATA_ARRAY_ORDER attribute must be unique among the relationship instances.

D.25 CDD$UNIQ_DAC_SEQ_NUM

For each instance of the CDD$DATA_AGGREGATE_CONTAINS relationship owned by a given CDD$DATA_AGGREGATE entity, the value of the CDD$DATA_SEQUENCE_NUMBER attribute must be unique.

D.26 CDD$UNIQ_DEPEND_ID

If a CDD$DATA_VALUE entity owns more than one instance of the CDD$DATA_VALUE_DEPENDS_ON relationship, each relationship instance must have a unique value of the CDD$DATA_VALUE_DEPENDENCY_ID attribute.

D.27 CDD$UNIQ_DOAC_SEQ_NUM

For each instance of the CDD$DATA_OVERLAY_AGG_CONTAINS relationship owned by a given CDD$DATA_OVERLAY_AGGREGATE entity, the value of the CDD$DATA_SEQUENCE_NUMBER attribute must be unique.

D.28 CDD$UNIQ_DOC_SEQ_NUM

For each instance of the CDD$DATA_OVERLAY_CONTAINS relationship owned by a given CDD$DATA_OVERLAY entity, the value of the CDD$DATA_SEQUENCE_NUMBER attribute must be unique.

D.29 CDD$UNIQ_INST_PATH

If a CDD$DATA_INSTANCE entity owns more than one instance of the CDD$DATA_INSTANCE_PATH relationship, each relationship instance must have a unique value of the CDD$DATA_INSTANCE_PATH_STEP attribute.

E
Literal Values

This appendix lists Oracle CDD/Repository literal constants and their values.

E.1 General-Purpose Buffer Tags

These buffer tags are unsigned bytes. Table E–1 shows the value for each buffer tag.

Table E–1 Buffer Tags and Values

Buffer Tag	Tag Value
cdd$k_action	76
cdd$k_all	11
cdd$k_and	92
cdd$k_begin	1
cdd$k_buf_high	111
cdd$k_buf_low	1
cdd$k_cdd	109
cdd$k_default_name	105
cdd$k_dictionary_type	108
cdd$k_directory_name	15
cdd$k_directory_name_list	104
cdd$k_element	103
cdd$k_element_handle	29
cdd$k_end	5
cdd$k_entity	2
cdd$k_entity_handle	82

(continued on next page)

Table E–1 (Cont.) Buffer Tags and Values

Buffer Tag	Tag Value
cdd$k_eoc	8
cdd$k_eql	84
cdd$k_eql_one	90
cdd$k_error	111
cdd$k_exists	96
cdd$k_geq	87
cdd$k_gtr	86
cdd$k_leq	89
cdd$k_literal	20
cdd$k_lss	88
cdd$k_missing	25
cdd$k_nad	110
cdd$k_neq	85
cdd$k_no_member	102
cdd$k_no_order	64
cdd$k_not	91
cdd$k_not_valid	26
cdd$k_op_high	97
cdd$k_op_low	84
cdd$k_optional	63
cdd$k_or	93
cdd$k_ordering	45
cdd$k_query	21
cdd$k_related_name	106
cdd$k_relop_high	89
cdd$k_relop_low	84
cdd$k_size	107
cdd$k_type	30
cdd$k_unique	97

(continued on next page)

Table E–1 (Cont.) Buffer Tags and Values

Buffer Tag	Tag Value
cdd$k_using	13
cdd$k_value_list	83
cdd$k_within_exp	32
cdd$k_xor	94

E.2 Justification Flags

Justification flags are unsigned words. Table E–2 shows the value of each justification flag.

Table E–2 Justification Flags

Justification Flag	Tag Value
cdd$k_justified_left	1
cdd$k_justified_right	2
cdd$k_justified_center	3

E.3 Notice Types

Notice type values are unsigned longwords. Table E–3 shows the value of each notice type.

Table E–3 Notice Types

Notice Type	Tag Value
mcs_possibly_invalid	1
mcs_invalid	26
mcs_child_usage	2
mcs_new_version	4

E.4 Notice Action Flags

Notice action flags are unsigned words. Table E–4 shows the value for each notice action flag.

Table E–4 Notice Action Flag

Notice Action Flag	Tag Value
mcs_success	1
mcs_signal	2
mcs_block	3

E.5 Protection Bits in Access Control Lists

An unsigned longword of bits is used to specify privileges in an access control list. Table E–5 shows the value for each privilege.

Table E–5 Protection Bits in Access Control Lists

Privilege	Tag Value
cdd$k_read_priv	0
cdd$k_write_priv	1
cdd$k_modify_priv	2
cdd$k_erase_priv	3
cdd$k_show_priv	4
cdd$k_define_priv	5
cdd$k_change_priv	6
cdd$k_delete_priv	7
cdd$k_control_priv	8
cdd$k_oper_priv	9
cdd$k_admin_priv	10
cdd$m_read_priv	1
cdd$m_write_priv	2
cdd$m_modify_priv	4

(continued on next page)

Table E–5 (Cont.) Protection Bits in Access Control Lists

Privilege	Tag Value
cdd$m_erase_priv	8
cdd$m_show_priv	16
cdd$m_define_priv	32
cdd$m_change_priv	64
cdd$m_delete_priv	128
cdd$m_control_priv	256
cdd$m_oper_priv	512
cdd$m_admin_priv	1024
cdd$m_all_priv	2047

Index

A

B

C

D

E

F

I

J

L

M

N

T

U

V

W