IBM Distributed Computing Environment Version 3.1
for AIX and Solaris:

Application Development Guide – Directory Services

IBM Distributed Computing Environment Version 3.1
for AIX and Solaris:

Application Development Guide – Directory Services

Note

Before using this document, read the general information under “Appendix. Notices” on page 259.

First Edition (August 1999)

This edition applies to Version 3.1 of *IBM Distributed Computing Environment for AIX and Solaris* and to all subsequent releases and modifications until otherwise indicated in new editions or technical newsletters.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address below.

IBM welcomes your comments. Send your comments to the following address:

International Business Machines Corporation
Department VLXA
11400 Burnet Road
Austin, Texas
78758

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Contents

Figures

Tables

Preface

The *IBM DCE Version 3.1 for AIX and Solaris: Application Development Guide* provides information on how to access the DCE Directory Service on an IBM® and Solaris operating system to share information and resources between cells. The Directory Service primarily includes three components:

- DCE Cell Directory Service (CDS)
- DCE Global Directory Service (GDS) (not supported for this release)
- X/Open Directory Service (XDS) and X/Open OSI-Abstract-Data Manipulation (XOM) Interfaces.

The *IBM DCE Version 3.1 for AIX and Solaris: Application Development Guide* provides information about how to program the application programming interfaces (APIs) provided for each OSF® Distributed Computing Environment (DCE) component.

Audience

This guide is written for application programmers with AIX, Solaris, or UNIX® operating system and C language experience who want to develop and write applications to run on DCE. It does not assume that you have prior knowledge of, or experience with, designing and writing distributed applications using the Open Software Foundation's (OSF) Distributed Computing Environment (DCE) services. Ideally, you should be able to perform the following:

- Edit, browse, and copy AIX and Solaris files
- Print files
- Write, compile, link, debug, and run C programs on AIX and Solaris.

A good working knowledge and understanding of the following would also be helpful:

- Structured programming techniques
- Computer communications over a network using Transmission Control Protocol (TCP) and User Datagram Protocol (UDP)
- Concepts behind a distributed application.

Some exposure to the AIX, Solaris, or UNIX operating systems is helpful but not essential to use this guide.

Purpose

The purpose of this guide is to assist programmers in developing applications that use DCE. After reading this guide, you should be able to program the Application Programming Interfaces provided for each DCE component.

Document Usage

The *IBM DCE Version 3.1 for AIX and Solaris: Application Development Guide* consists of three books, as follows:

- *IBM DCE Version 3.1 for AIX and Solaris: Application Development Guide—Introduction and Style Guide*

- *IBM DCE Version 3.1 for AIX and Solaris: Application Development Guide—Core Components*
- *IBM DCE Version 3.1 for AIX and Solaris: Application Development Guide—Directory Services*
  - DCE Directory Service
  - CDS Application Programming
  - XDS/XOM Application Programming
  - XDS/XOM Supplementary Information

Related Documents

For additional information about the Distributed Computing Environment, refer to the following documents:

- *IBM DCE Version 3.1 for AIX and Solaris: Application Development Guide*
- *IBM DCE Version 3.1 for AIX and Solaris: Introduction to DCE*
- *IBM DCE Version 3.1 for AIX and Solaris: Administration Commands Reference*
- *IBM DCE Version 3.1 for AIX and Solaris: Application Development Reference*
- *IBM DCE Version 3.1 for AIX and Solaris: Administration Guide*
- *OSF DCE GDS Administration Guide and Reference*
- *OSF DCE/File-Access Administration Guide and Reference*
- *OSF DCE/File-Access User's Guide*
- *IBM DCE Version 3.1 for AIX and Solaris: Problem Determination Guide*
- *OSF DCE Testing Guide*
- *OSF DCE/File-Access FVT User's Guide*
- *Application Environment Specification/Distributed Computing*
- *OSF DCE Technical Supplement*
- *IBM DCE Version 3.1 for AIX: Release Notes*
- *IBM DCE Version 3.1 for Solaris: Release Notes*

Typographic and Keying Conventions

This guide uses the following typographic conventions:

Bold	**Bold** words or characters represent system elements that you must use literally, such as commands, options, and pathnames.
Italic	*Italic* words or characters represent variable values that you must supply. *Italic* type is also used to introduce a new DCE term.
`Constant width`	Examples and information that the system displays appear in `constant width` typeface.
[ ]	Brackets enclose optional items in format and syntax descriptions.
{ }	Braces enclose a list from which you must choose an item in format and syntax descriptions.
|	A vertical bar separates items in a list of choices.
< >	Angle brackets enclose the name of a key on the keyboard.
...	Horizontal ellipsis points indicate that you can repeat the preceding item one or more times.

dcelocal
The OSF variable *dcelocal* in this document equates to the AIX and Solaris value **/opt/dcelocal**.

dceshare
The OSF variable *dceshare* in this document equates to the AIX and Solaris value **/opt/dcelocal**.

This guide uses the following keying conventions:

<Ctrl-x> or ˆ x
The notation **<Ctrl-x>** or **ˆ x** followed by the name of a key indicates a control character sequence. For example, **Ctrl-C** means that you hold down the control key while pressing **<C>**.

<Enter>
The **<Enter>** notation refers to the key on your terminal or workstation that is labeled with the word Enter or Return, or with a left arrow.

Entering commands
When instructed to enter a command, type the command name and then press the **<Enter>** key. For example, the instruction ″Enter the IDL command″ means that you type the IDL command and then press the **<Enter>** key.

Terminology Used in This Book

Although every attempt has been made to conform to Systems Application Architecture (SAA) terminology guidelines, you must keep in mind that the DCE technology has been developed from the UNIX environment.

Notes:

1. Throughout this document, the terms *API, call,* and *routine* all refer to the same AIX and Solaris application programming interface that is referenced. For example, **rpc_binding_free( )** API, **rpc_binding_free call( )**, and **rpc_binding_free( )** routine, all refer to the same **rpc_binding_free( )** API.
2. Throughout this document, all references to individual DCE components (such as RPC) refer to that component with the AIX and Solaris product. For example, references to RPC, DCE RPC, and IBM DCE for AIX and Solaris RPC all refer to the RPC component of IBM DCE for AIX and Solaris.

Pathnames of Directories and Files in DCE Documentation

For a list of the pathnames for directories and files referred to in this guide, see the *IBM DCE Version 3.1 for AIX and Solaris: Administration Guide—Introduction* and the *OSF DCE Testing Guide*.

Part 1. DCE Directory Service

Chapter 1. DCE Directory Service Overview

This chapter provides an overview of the *IBM DCE Version 3.1 for AIX and Solaris: Application Development Guide—Directory Services* for application programmers. The chapter begins with a description of this guide. It then introduces DCE Directory Service concepts, following which the structure of DCE names and the DCE namespace are described. The chapter then provides an overview of the programming interfaces used to access the DCE Directory Service.

Introduction to This Guide

This guide describes how application developers can access the DCE Directory Service. From the application programmer's perspective, the directory service has three main parts: the DCE Cell Directory Service (CDS), the DCE Global Directory Service (GDS), and the X/Open Directory Service (XDS) and X/Open OSI-Abstract-Data Manipulation (XOM) programming interfaces. GDS is not supported for this release of DCE. See the DCE for AIX Version 1.3 publications if you need information on GDS.

This product contains support for the XDS/XOM API over CDS. Although the documentation contains numerous references to the functions provided by the Global Directory Services (GDS), only the XDS/XOM API over CDS functionality is shipped with this product. The Global Directory Service functionality is available as a separate product on AIX Version 3.2.5. It is not supported on AIX Version 4.1 and greater.

Use of This Guide

Before reading this guide, you should read the *IBM DCE Version 3.1 for AIX and Solaris: Introduction to DCE* . It contains overviews, along with illustrations, of all the DCE components and of DCE as a whole, as well as introductions necessary to fully understand what is described here. Next, read this section in its entirety.

If you do not find the information you need in either this guide or the *IBM DCE Version 3.1 for AIX and Solaris: Application Development Reference*, see the *IBM DCE Version 3.1 for AIX and Solaris: Administration Guide* and the *IBM DCE Version 3.1 for AIX and Solaris: Administration Commands Reference*. For example, information about the CDS as a separate component is found in the *IBM DCE Version 3.1 for AIX and Solaris: Administration Guide*. Although the DCE Security Service is documented in the *IBM DCE Version 3.1 for AIX and Solaris: Application Development Guide*, some information of interest to programmers (such as adding new principals to the registry database) is also found in the *IBM DCE Version 3.1 for AIX and Solaris: Administration Guide*.

Directory Service Tools

CDS has commands that allow system administrators to inspect and alter the contents of the directory (service). This can be useful when developing applications that access the DCE namespace.

For information on the CDS control program (**cdscp**), see the *IBM DCE Version 3.1 for AIX and Solaris: Administration Guide—Core Components*.

Using the DCE Directory Service

The DCE Directory Service can be used in many ways. It is used by the DCE services themselves to support the DCE environment. For example, cells are registered in the global part of the directory service, enabling users from different cells to share information and resources (you will need an AIX Version 1.3 system as part of your cell to perform this function if you are using X.500 cell names and IBM's version of GDS).

The directory service is also useful to DCE applications. The client and server sides of an application can use it to find each other's locations. The directory service can also be used to store information that must be made available in a globally accessible, well-known place.

For example, one DCE application could be a print service consisting of a client side application that makes requests for jobs to be printed, and a server-side application that prints jobs on an available printer. The directory service could be used as a central place where the print clients could look up the location of a print server. It could also be used to store information about printers; for example, what type of jobs a printer can accept and whether it is currently up or down and lightly or heavily loaded.

In some ways, a directory service can be used in the same way that a file system has traditionally been used; that is, for containing globally accessible information in a well-known place. An example is the use of configuration information stored in files in a UNIX **/etc** directory.

However, the directory service differs in important ways. It can be replicated so that information is available even if one server goes down. Replicas can be kept automatically up-to-date so that, unlike multiple copies of a file on different machines, the information in the replicas of the directory service can be kept current without manual intervention.

The directory service can also provide security for data that is kept in a globally accessible place. It supports access control lists (ACLs) that provide fine-grained control over who is able to read, modify, create, and perform other operations on its data.

As you learn about the directory service and how to access it, think about the ways in which your application can best take advantage of the services it provides.

DCE Directory Service Concepts

This section provides a description of DCE Directory Service concepts that are important to application developers. The following concepts are intended to convey general definitions that are applicable to the directory service as a whole rather than specific to a particular directory service component. For more detailed definitions, see the glossary in the *IBM DCE Version 3.1 for AIX and Solaris: Introduction to DCE*.

- DCE namespace

  The DCE namespace is the collection of names in a DCE environment. It can be made up of several domains, in which different types of servers own the names in different parts of the namespace. Typically, there are two high-level, or global, domains to a DCE namespace: the GDS namespace and the Domain Name

System (DNS) namespace. At the next level is the CDS namespace, with names contained in the cell's CDS server. A DCE environment always contains a cell namespace, which is implemented by CDS. Parts of the DCE namespace may not be contained in any of the directory services; for example, the DFS (Directory File Service) namespace, also called the filespace, contains the names of files and directories in DFS, and the security namespace contains principals and groups contained in the security server.

The term *DCE namespace* is used when referring to names, but not the information associated with them. For example, it would include the name of a printer in the directory service, but not its associated location attribute, and it would include the name of a DFS file, but not its contents.

- Cell namespace

  All of the names found in a single DCE cell constitute the cell's namespace. This includes names managed by the cell's CDS server and security server, names in the cell's DFS if it has one, and any other names that reside within a particular cell.

- Hierarchy

  The DCE namespace is organized into a hierarchy; that is, each name except the global root has a parent node and may itself have child nodes or leaves. The leaves are called objects or entries, and, in the CDS and DFS namespace, the nodes are called directories.

- Directory

  The word *directory* has two meanings, which can be differentiated by their context. The first is the node of a hierarchy as mentioned in the previous definition. The second is a collection of objects managed by a directory service.

- Directory service

  A directory service is software that manages names and their associated attributes. A directory service can store information, be queried about information, and be requested to change information. IBM DCE contains Cell Directory Service (CDS) and interacts with DNS (not part of DCE).

- Junction

  A junction is a point in the DCE namespace that contains binding information that enables a client to connect to a service outside of CDS. It is a well-defined point in the DCE namespace after which a server other than CDS controls. For example, the point where the DFS entries are *mounted* into a CDS namespace is a junction. DCE also has junctions between the global directory services and CDS, and between CDS and the DCE Security Service.

- Object

  The word *object* can have two meanings, depending on the context. Sometimes it means an entry in a directory service. Sometimes it means a real object that an entry in a directory service describes, such as a printer. In the context of XDS/XOM, the requested data is returned to the application in one or more *interface objects*, which are data structures that the application can manipulate.

- Entry

  An entry is a unit of information in a directory service. It consists of a name and associated attributes. For example, an entry could consist of the name of a printer, its capabilities, and its network address.

  – Class

    In GDS, each entry has a class associated with it. The class determines what type of entry it is and what attributes may be associated with it. Class is optional in CDS.

  – Link

    A link is one type of object class. This type of object is a pointer to another object; it is similar to a soft link in a UNIX file system. A CDS link is similar to a GDS alias.

- Attribute

  If an object is like a complex data structure, then its attributes are analogous to the separate member fields within that structure. Some of an object’s attributes may be of significance only to the directory service that manages it. With attributes such as these, a directory service implements objects that contain various kinds of data about the directory itself, thus enabling the service to organize the entries into a meaningful structure. For example, directory objects can contain attributes whose values are other directory objects (called child directories or subdirectories) in the directory. Or link objects can contain attributes whose values are the names and internal identifiers of other directory entries, making a link object’s entry name an alias of the other object to which its attributes indirectly refer.

  – Type

    Every attribute is characterized as being of a certain type. The attribute is used to hold a certain kind of data, such as a zip code or the name of a cat. Entries can also be classified by type; for entries, the term used is *class*.

  – Value

    An attribute can have one or more values.

- Object identifier

  Directory attributes are uniquely identified by object identifiers (OIDs), which are administered by the International Organization for Standardization (ISO). In GDS, OIDs are also used to identify object classes. When it creates new attribute types, an application is responsible for tagging them with new, properly allocated OIDs (see your directory service administrator for OID assignments). In CDS, attribute types are identified by strings, that can be representations of OIDs. The **cds_attributes( )** file contains the string to OID mappings.

- Name

  A DCE name corresponds to an entry in some service participating in the DCE namespace, usually a directory service (see DCE Cell Namespace in the *IBM DCE Version 3.1 for AIX and Solaris: Administration Guide*).

  – Global name

    A global name is a name that contains a path through one of the global namespaces (GDS or DNS).

  – Local name

    A local name is a name that uses the cell prefix **/.:** to indicate the cell name and therefore does not have a specific path through a global namespace. The entry for a local name is always contained in the local cell.

- Access control list

  Access to DCE namespace entries is determined by lists of entities that are attached through the DCE Security Service to both the entries and the objects when they are created. The lists, called access control lists (ACLs), specify the privileges that an entity or group of entities has for the entry the ACL is associated with. The security service provides servers with authenticated identification of every entity that contacts them; it is then the server’s responsibility to check the ACL attached to the object that the potential client wants to access, and perform or refuse to perform the requested operation on the basis of what it finds there. The ACLs are checked using security service library routines.

Objects in the GDS namespace have ACLs associated with them, but they are not security service ACLs.

- Replication

  The DCE Directory Service can keep replicas (copies) of its data on different servers. This means that, if one server is unavailable, clients can still obtain information from another server.

- Caching

  Both the CDS and GDS components of the directory service support caching of data on the client machine. When a client requests a piece of data from the directory service for the first time, the information must be obtained over the network from a server. However, the data can then be cached (stored) on the local machine, and subsequent requests for the same data can be satisfied more quickly by looking in the local cache instead of sending a request over the network. You need to be aware of caching because in some cases you will want to bypass the cache to ensure that the data you obtain is as up-to-date as possible.

Structure of DCE Names

The following subsections describe the structure of the names found in a DCE environment. DCE names can consist of several different parts, which reflect the federated nature of the DCE namespace. A DCE name has some combination of the following elements. They must occur in this order, but most elements are optional.

- Prefix
- GDS cell name or DNS cell name
- GDS name or CDS name
- Junction
- Application name

A DCE name can be represented by a string that is a readable description of a specific entry in the DCE namespace. The name is a string consisting of a series of elements separated by / (slash). The elements are read from left to right. Each consecutive element adds further specificity to the entry being described, until finally one arrives at the rightmost element, which is the simple name of the entry itself. Thus, in appearance, DCE names are similar to UNIX filenames.

In the discussion that follows, a DCE name *element* is the single piece of a name string enclosed between a consecutive pair of slashes. For example, consider the following string:

```
/.../C=US/O=OSF/OU=DCE/hosts/abc/self
```

In it, the following two substrings are both elements:

```
O=OSF
```

```
abc
```

The entire name contains (counting the `...` element) a total of seven elements.

In GDS, an element is called a relative distinguished name (RDN) and the entire name is called a distinguished name (DN). In the preceding example, the attribute type **O** stands for the Organization type OID, which is 2.5.4.10.

DCE Name Prefixes

The leftmost element of any valid DCE name is a root prefix. The appearance and meaning of each is as follows:

/... This is the *global root*. It signifies that the immediately following elements form the name of a global namespace entry. Usually, the entry's contents consist of binding information for a DCE cell (more specifically, for some CDS server in the cell), and the name of the global entry is the name of the cell.

/.: This is the *cell root*. It is an alias for the global prefix plus the name of the local cell; that is, the cell in which the prefix is being used. It signifies that the immediately following elements taken together form the name of a cell namespace entry in the local cell.

/: This is the *filespace root*. It is an alias for the global prefix, the name of the local cell, and the DFS junction.

DCE also supports a junction into the security service namespace, but there is no alias for this junction.

A prefix by itself is a valid DCE name. For example, you can list the contents of the **/.:** directory to see the top-level entries in the CDS namespace, and you can use a file system command to list the contents of the **/:** directory to see the top-level entries in the filespace.

Names of Cells

After the global root prefix, the next section of a DCE name contains the name of the cell in which the object's name resides. The name of a cell can be expressed as either a GDS name or a DNS name, depending on which global directory service (GDS or DNS) the cell is registered in. The following subsections provide examples.

GDS Cell Names

GDS elements always consist of a substring. Where an abbreviation or acronym in capital letters is followed by a = (equal sign), that is followed by a string value. These substrings represent pairs of attribute types and attribute values.

For example, consider the following global DCE name:

```
/.../C=DE/O=SNI/OU=DCE/subsys/druecker/docs
```

In it, the *attribute*= *value* form of the leftmost elements after the **/...** indicates that the global part of the name is a GDS namespace entry, and that it ends after the `OU=DCE` element; therefore, the rest of the name is in the **/.../C=DE/O=SNI/OU=DCE** cell.

DNS Cell Names

If DNS is used as the global directory, a global name has a form like the following:

```
/.../cs.univ.edu/subsys/printers/docs
```

where the single element

```
cs.univ.edu
```

is the cell name; that is, the cell's name in the DNS namespace.

Discovering Your Local Cell's Name

A DCE cell consists of the machines that are configured into it; each DCE machine belongs to one DCE cell. Your local cell is therefore the cell to which the machine you are using belongs. Depending on the DCE name you are using, you can access your own cell or other (foreign) cells. If the name you are accessing is global, then its cell is explicitly named. If the name begins with the local cell prefix, then you are accessing a name within your local cell. You can find out what cell you are in by calling the **dce_cf_get_cell_name( )** function.

Using the global directory services, applications can access resources and services in foreign cells; however, applications most frequently use resources from their local cell. If this is not the case, the cell boundaries may not have been well chosen.

CDS Names

After the cell name or cell root prefix, the next part of a DCE name is often a CDS name. For example, consider the following name:

```
/.../C=DE/O=SNI/OU=DCE/subsys/druecker/docs
```

The CDS part of this name is

```
/subsys/druecker/docs
```

Another example is the name

```
/.../cs.univ.edu/subsys/printers/docs
```

In this name, the CDS part is

```
/subsys/printers/docs
```

The following strings show equivalent names that use the cell root prefix, assuming that the name is used from within the **/.../C=DE/O=SNI/OU=DCE** and **/.../cs.univ.edu** cells, respectively:

```
/.:/subsys/druecker/docs
/.:/subsys/printers/docs
```

GDS Names

Some names fall entirely in the GDS namespace. These names are pure X.500 (and therefore GDS) names, since each element consists of a type and an attribute. The entries for these names are contained in a GDS server. The following is an example of a pure GDS name:

```
/.../C=US/L=Cambridge/CN=Kilroy
```

Junctions in DCE Names

Some junctions are implied in a DCE name; others can be seen. There is an implied junction between the global prefix and either GDS or DNS. It occurs after the global prefix. The junction between either of the global namespaces and the local cell namespace is also implied. It occurs after the cell name. The junction between the local cell namespace and either the DFS namespace or the security namespace is shown by the symbol **/fs** or **/sec**, respectively. The following are examples of visible junctions in DCE names:

```
/.:/fs/usr/snowpaws
/.../dce.osf.org/sec/principal/ziggy
```

Application Names

The part of a DCE name that occurs after a junction into a DCE application is the application name. DFS and security names are the currently supported examples; in the future, application programmers may also be able to create junctions in the namespace.

DFS names occur after the DFS junction. They are typeless and resemble UNIX file system names. After the global and CDS parts of a DFS name have been resolved by the appropriate directory services, the rest of the DFS name is handled within DFS. In the equivalent examples that follow, **/usr/snowpaws** is the DFS part of the DCE name:

```
/.../dce.osf.org/fs/usr/snowpaws
/.:/fs/usr/snowpaws
/:/usr/snowpaws
```

Security names are similar to DFS names; first the parts of the name within the DCE Directory Service are resolved, then the rest of the name is handled by the security service. The entry is contained in the security registry database. Consider the following:

```
/.:/sec/principal/ziggy
```

In this example, the security part of the DCE name is **/principal/ziggy**.

The Federated DCE Namespace

The DCE namespace is a single hierarchy of names, but the names can be contained in many different services. Because several services cooperate to make the DCE namespace, it is a federated namespace.

Figure 1 on page 11 shows a typical DCE namespace and the different services in which names reside.

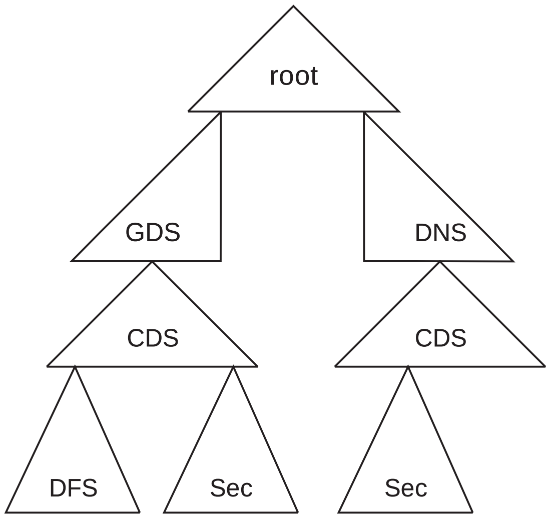


Figure 1. A Federated DCE Namespace

The following sections describe the different domains of the DCE namespace.

The GDS Namespace

This section provides a brief overview of the main characteristics of the GDS namespace regarded apart from the XDS interface used to access it.

In a GDS name such as

```
/.../C=US/O=OSF/OU=DCE
```

the `C=US` and `O=OSF` elements do not refer to directory entries that are fundamentally different from the one represented by `OU=DCE`, unlike in CDS or the UNIX file system.

Thus, in the name string

```
/C=US/O=OSF/OU=DCE
```

the element `C=US` refers to a one-level-down country entry whose value is `US`, then to a two-levels-down organization entry whose value is `OSF`, and then to a three-levels-down organization unit entry whose value is `DCE`. Concatenating these elements results in a valid path of entries from the directory root to the `DCE` entry. The entry itself is the namespace sign to a GDS directory object that contains binding information for the **/.../C=US/O=OSF/OU=DCE** cell.

The GDS Schema

The schema defines the shape and format of entries in the GDS directory. It contains four types of rules, which describe the following:

- The legal hierarchy of entries. What entries may be subordinate to other entries. These rules are what prevents, for example, countries from being subordinate to states.
- The allowable object classes, the mandatory and optional attributes of entries, and which attributes are the naming attributes.

- The allowable attribute types, associating a unique OID and an attribute syntax with each attribute type.
- The syntaxes of attributes that describe what attribute values look like, such as strings, numbers, or OIDs.

By installing the proper schema, an entry of any particular object class can have the two attributes needed to identify a cell.

The CDS Namespace

The CDS namespace is the part of the DCE namespace that resides in the local cell’s CDS. DCE itself is made up of components that, like the applications that use them, are distributed client/server applications. These components rely on CDS to make themselves available as services to DCE applications. This requires that the structure of the cell namespace be stable, known, and have parts that are not alterable by casual users or applications.

The CDS Schema

The cell namespace’s hierarchy model is different from the GDS model, and the CDS rules do not enforce any particular model; CDS allows entries containing any kind of data to be created anywhere in the namespace. Thus, CDS offers a free-form namespace in which entries and directories can be organized as desired, and in which any entry or directory can contain any attributes. The CDS administrator can create additional directories, and applications can add object entries as needed; applications *cannot* create CDS directory entries. Because of this, and because the cell namespace is so important to the operation of the cell, application developers and system administrators have more responsibility in planning and regulating their use of it.

The cell namespace has a structure similar to that of a UNIX file system. The CDS namespace is a tree of entries that grows from the root downward. The name entries are organized under directory entries, which can themselves be subentries of other directories. The cell root (represented by the prefix **/.:**) can be thought of as the location you get when you dereference the cell’s global name. New directories and new entries within the directories can be added anywhere in the tree, subject to the restrictions mentioned previously.

CDS Entries and CDS Attributes

There are three different kinds of CDS entries that are of significance to application programmers, as follows:

- Object
- Soft link
- Directory

The object entries are the most primitive form. These are where data is stored. Directory entries contain other entries (that is, can have children) just like UNIX file system directories. Soft link entries are essentially alias names for other directory or object entries. Only object entries can be created by applications; soft links and directories have to be created and manipulated with the **cdscp** command.

Thus, any CDS entry is defined as a directory, a soft link, or an object entry by the presence of a certain combination of attributes belonging to that kind of entry. You can use the **cdscp** command to get a display of all the attributes of any CDS entry.

The term *attribute* as applied to namespace entry objects has roughly the same meaning in CDS and GDS. The main difference is that CDS does not restrict or control the combinations of attributes attached to entries written in its namespace.

Other Namespaces

For information about names contained in the DFS namespace (the filespace) and the security namespace, refer to the chapters on those components in this guide.

Programming Interfaces to the DCE Directory Service

The following two subsections describe two programming interfaces for accessing the DCE Directory Service.

The XDS Interface

The main programming interface to all services within the directory service is XDS/XOM, as defined by X/Open. The calls correspond to the X.500 service requests, including Read, List (enumerate children), Search, Add Entry, Modify Entry, Modify RDN, and Remove Entry. XDS uses XOM to define and manipulate data structures (called *objects*) used as the parameters to these calls, and used to describe the directory entries manipulated by the calls. XOM is extremely flexible, but also somewhat complex. The interfaces are used in different ways, depending on which underlying directory service is being addressed. For example, CDS entries are typeless, but GDS entries are typed. This difference is reflected in the use of the interface.

The RPC Name Service Interface

The DCE Remote Procedure Call (RPC) facility supports an interface to the directory service that is specific to RPC and is layered on top of directory service interfaces; it is called the Name Service Independent (NSI) interface. NSI can manipulate three object classes — entries, groups, and profiles — which were created to store RPC binding information. NSI data is stored in CDS. Programming using this interface is discussed in the *IBM DCE Version 3.1 for AIX and Solaris: Application Development Guide—Core Components* and *IBM DCE Version 3.1 for AIX and Solaris: Application Development Guide—Introduction and Style Guide* volumes.

Namespace Junction Interfaces

For information about programming interfaces to names that occur in namespace junctions, see the documentation for that component.

Part 2. CDS Application Programming

This section describes DCE Directory Service application programming in the CDS namespace. It describes the contents of the CDS namespace, where applications should put their data, and what the valid CDS characters and names are. It also describes how to use the XDS programming interface to access data in the CDS namespace.

Chapter 2. Programming in the CDS Namespace

This chapter provides information about writing applications that use the XDS/XOM interface to access the portion of the DCE namespace contained in CDS.

The XDS/XOM interface provides generalized access to CDS. However, if you only need to use CDS to store information related to RPC (for example, storing the location of a server so that clients can find it), you should use the NSI interface of DCE RPC. NSI implements RPC-specific use of the namespace. For information on using RPC NSI, see the *IBM DCE Version 3.1 for AIX and Solaris: Application Development Guide—Core Components*.

For information on the details of accessing the CDS namespace through the XDS/XOM interface, see “Chapter 3. XDS and the DCE Cell Namespace” on page 35.

Initial Cell Namespace Organization

The following subsections describe the organization of a cell’s namespace after it has initially been configured. (For more information on configuring a cell, see the *IBM DCE Version 3.1 for AIX and Solaris: Administration Guide*.)

Every DCE cell is set up at configuration with the basic namespace structure necessary for the other DCE components to be able to find each other and to be accessible to applications. The vital parts of the namespace are protected from being accessed by unauthorized entities by ACLs that are attached to the entries and directories.

Figure 2 on page 18 shows what the cell namespace looks like after a cell has been configured and before any additional directories or entries have been added to it by system administrators or applications. In the figure, ovals represent directories, rectangles represent simple entries, circles represent soft links, and triangles represent namespace junctions.

All of the simple entries shown in the figure are created for use with RPC NSI routines; that is, they all contain server-binding information and exist to enable clients to find servers. These are referred to as *RPC entries*.

Note that only the name entries (those in boxes) and junction entries (those in triangles) are RPC entries. The directories (entries indicated by ovals) are normal CDS directories.

Some of the namespace entries in the figure are intended to be used (if desired) directly by applications; namely, **/.:/cell-profile**, **/.:/lan-profile**, and, through the **/:** soft link alias, **/.:/fs**. The `self` and `profile` name entries under `hosts` also fall into this category. Others, such as those under **/.:/subsys/dce**, are for the internal use of the DCE components themselves.

Each of the entries is explained in detail in the following subsections. See the *IBM DCE Version 3.1 for AIX and Solaris: Administration Guide* for detailed information on the contents of the initial DCE cell namespace.

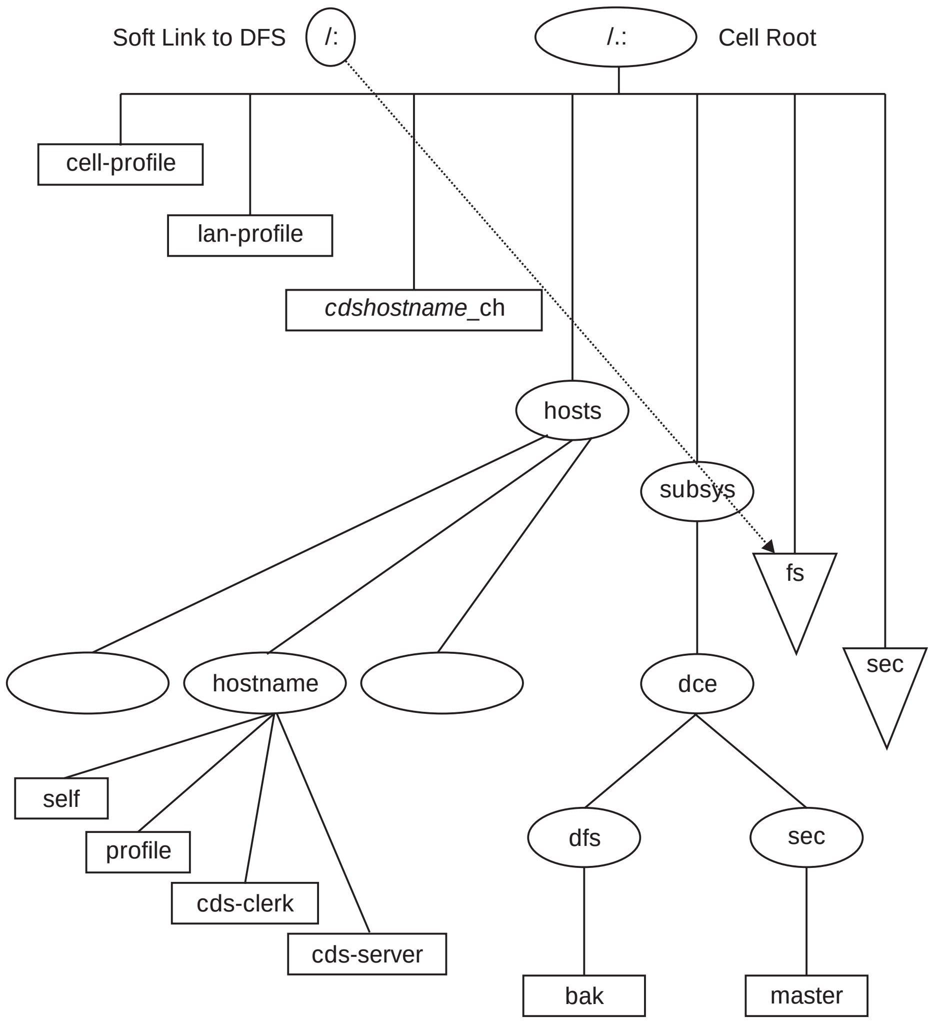


Figure 2. The Cell Namespace After Configuration

The Cell Profile

The **/.:/cell-profile** entry is an RPC profile entry that contains the default list of namespace entries to be searched by clients trying to bind to certain basic services. An RPC profile is a class of namespace entry used by the RPC NSI routines. When a client imports bindings from such an entry, it imports, through the profile, from an ordered list of RPC entries containing appropriate bindings. The list of entries is keyed by their interface universal unique identifiers (UUIDs) so that only bindings to servers offering the interface sought by the client are returned. The entries listed in the profile exist independently in the namespace, and can be separately accessed in the normal way. The profile is simply a way of organizing clients' searches.

The main purpose of `cell-profile` is as a path of last resort for prospective clients. All other profile entries in the cell namespace are required to have the `cell-profile` entry in *their* entry lists so that, if a client exhausts a particular profile's list of entries, it tries those in `cell-profile`.

The LAN Profile

The **/.:/lan-profile** entry is a local area network (LAN)-oriented default list of services' namespace entries that is used when servers' relative positions in the network topography are of importance to their prospective clients.

The CDS Clearinghouse

The **/.:/***cdshostname*`_ch` entry is the namespace entry for *cdshostname*'s clearinghouse, where *cdshostname* is the name of the host machine on which a CDS server is installed.

A *clearinghouse* is the database managed by a CDS server; it is where CDS directory replicas are physically stored. For more information about clearinghouses, see the *IBM DCE Version 3.1 for AIX and Solaris: Administration Guide*. All clearinghouse namespace entries reside at the cell root, and there must be at least one in a DCE cell. The first clearinghouse's name must be in the form shown in Figure 2 on page 18, but additional clearinghouses can be named as desired.

The Hosts Directory

The **/.:/hosts** entry is a directory containing entries for all of the host machines in the cell. Each host has a separate directory under `hosts`; its directory has the same name as the host machine. Four entries are created in each host's directory:

- `self`

  This entry contains bindings to the host's DCE daemon (`dced`), which is responsible for, among other things, dynamically resolving the partial bindings that it receives in incoming RPCs from clients attempting to reach servers resident on this host.
- `profile`

  This entry is the default profile entry for the host. This profile contains in its list of entries at least the **/.:/cell-profile** entry described in "The Cell Profile" on page 18.
- `cds-clerk`

  This entry contains bindings to the host's resident CDS clerk.
- `cds-server`

  This entry contains bindings to a CDS server.

The Subsystems Directory

The **/.:/subsys** entry is the directory for subsystems. Subdirectories below **subsys** are used to hold entries that contain location-independent information about services, particularly RPC binding information for servers.

The `dce` directory is created below **/.:/subsys** at configuration. This directory contains directories for the DCE Security Service and Directory File Service (DFS) components. The functional difference between these two directories and the `fs` and `sec` junctions described on page "The DFS and DCE Security Service Junctions" on page 20 is that the latter two entries are the access points for the components' special databases, whereas the directories under **subsys/dce** contain the services' binding information.

Subsystems that are added to DCE should place their system names in directories created beneath the **/.:/subsys** directory. Companies adding subsystems should conform to the convention of creating a unique directory below **subsys** by using their trademark as a directory name. Use these directories for storage of location-independent information about services. You should store server entries, groups and profiles for the entire cell in the directories below **subsys**. For example, International Air Freight-supplied subsystems should be placed in **/.:subsys/IAF**.

The /: DFS Alias

The entry **/:** is created and set up as a soft link to the **/.:/fs** entry, which is the DFS database junction. The name **/:** is equivalent to **/.:/fs**. Note, however, that the name **/:** is well-known, whereas the name **/.:/fs** is not, so using **/:** makes an application more portable. A CDS soft link entry is an alias to some other CDS entry. A soft link is created through the **cdscp** command. The procedure is described in the *IBM DCE Version 3.1 for AIX and Solaris: Administration Guide*.

The DFS and DCE Security Service Junctions

The **/.:/fs** entry is the DFS junction entry. This is the entry for a server that manages the DFS file location database.

The **/.:/sec** entry is the DCE Security Service junction entry. This is the entry for a server that manages the security service database (also called the registry database).

The **/.:/fs** and **/.:/sec** root entries in Figure 2 on page 18 are junctions maintained by DCE components. The **/.:/sec** junction is the security service's namespace of principal identities and related information. The DFS's fileset location servers are reached through the **/.:/fs** entry, making **/.:/fs** effectively the entry point into the cell's distributed file system.

Note that **/.:/sec** and **/.:/fs** are both actually RPC group entries; the junctions are implemented by the servers whose entries are members of the group entries. (See the *IBM DCE Version 3.1 for AIX and Solaris: Administration Guide* for further details on the security service and DFS junctions.)

Recommended Use of the CDS Namespace

CDS data is maintained in a consistently loose manner. This means that, when the writeable copy of a replicated name is updated, the read-only copies might not be updated for some period of time, and applications reading from those nonsynchronized copies can receive stale data. This is in contrast to distributed databases, which use multiphase commit protocols that prevent readers from accessing potentially stale or inconsistent data while the writes are being propagated to all copies of the data. It is possible to specifically request data from the master copy, which is guaranteed to be up-to-date, but replication advantages are then lost. This should only be done when it is important to obtain current data.

Storing Data in CDS Entries

Some CDS entries may contain information that is immediately useful or meaningful to applications. Other entries may contain RPC information that enables application clients to reach application servers; that is, binding handles for servers, which are stored and retrieved using the RPC NSI routines. In either case, the entry's name should be a meaningful identification label for the information that the entry contains. This is because the namespace entry names are the main clue that users and applications have to the available set of resources in the DCE cell. Using the CDS namespace to store and retrieve binding information for distributed applications is the function of DCE RPC NSI.

In general, applications can store data into CDS object entry attributes in any XDS-expressible form they wish. Refer to Table 6 on page 74 and Table 7 on page 75 for XDS-to-CDS data type translations. If you add new attributes to the **/opt/dcelocal/etc/cds_attributes** file, together with a meaningful CDS syntax (that is, a data type identifier) and name, then the attribute is displayed with the string representation of the OID by **cdscp show** commands when executed on objects containing instances of that attribute. If the attribute has not been added to **cds_attributes( )**, then the OID is displayed.

There are three main questions to consider when using CDS to store data through application calls to XDS:

1. `Where in the CDS namespace should the new entries be placed?`

   You are free to create new directories as long as you do not disturb the namespace's configured structure. Keep in mind that CDS directories must be created with the **cdscp** or **dcecp** command; they cannot be created by applications.

   Only two root-level directories are created at configuration: `hosts` and **subsys**. Applications should not add entries under the `hosts` tree; the host's default profile should instead be set up by a system administrator. The **subsys** directory is intended to be populated by directories (for example, **/.:/subsys/dce**) in which the servers and other components of independent vendors' distributed products are accessed. Thus, the typical cell should usually have a series of root-level CDS directories that represent a reasonable division of categories.

   One obvious division could be between entries intended for RPC use (that is, namespace entries that contain bindings for distributed applications), and entries that contain data of other kinds. On the other hand, it may be very useful to add supplementary data attributes to RPC entries in which various housekeeping or administrative data can be held. In this way, for example, performance data for printers can be associated with the print servers' name entries. You can either add new attributes to the server entries themselves, where, for example, the following is the name of a server entry that receives the new attributes:

   ```
   /.:/applications/printers/pr1
   ```

   Or you can change the subtree structure so that new *entries* are added to hold the data, the server bindings are still held in separate wholly RPC entries, and each group of entries is located under a directory named for the printer:

   ```
   /.:/applications/printers/pr1
   — directory
   /.:/applications/printers/pr1/server
   — server bindings
   /.:/applications/printers/pr1/stats
   — extra data
   ```

   In general, the same principals of logic and order that apply to the organization of a file system apply to the organization of a namespace. For example, server entries should *not* be created directly at the namespace root because this is the place for default profiles, clearinghouse entries, and directories.

Figure 3 illustrates some of the preceding suggestions, added to the initial configuration namespace structure shown in Figure 2 on page 18. In Figure 3, the vendor of the `xyz` subsystem has set up an `xyz` directory under **/.:/subsys** in which the system's servers are exported. This cell also has an **/.:/applications** directory in which the `printers` directory contains separate directories for each installed printer available on the system; the directory for `pr1` is illustrated in the figure. In the `pr1` directory, `server` is an RPC entry containing exported binding handles, and `stats` is an entry created and maintained through the XDS interface.

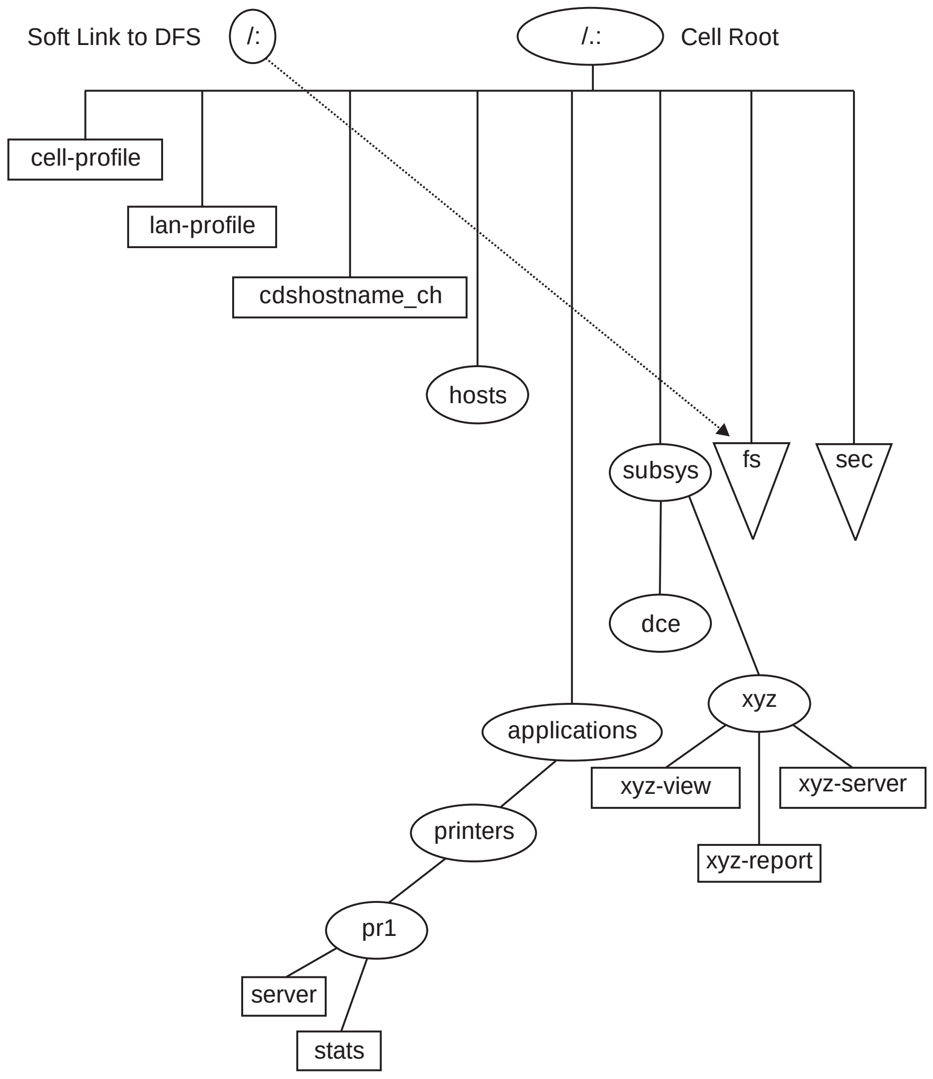


Figure 3. A Possible Namespace Structure

2. `How should the entries be constructed?`

   Because CDS allows you to add as many attributes as you wish to an object entry, it is up to you to impose some restraint in doing this. In view of the XDS overhead involved in reading and writing single CDS attributes, it makes sense to combine multiple related attributes under single entries (that is, in the same directory object) where they can be read and written in single calls to

ds_read( ) or **ds_modify_entry( )**. This way, for example, you only have to create one interface input object (to pass to **ds_read( )**) to read all the attributes, which you can do with one call to **ds_read( )**. You can then separate out the returned subobjects that you are interested in and ignore the rest. “Chapter 3. XDS and the DCE Cell Namespace” on page 35 contains detailed discussions of XDS programming techniques.

In any case, you should define object types for use in applications so that namespace access operations can be standardized and kept efficient. A CDS object type consists of a specific set of attributes that belong to an object of that type, with no other attributes allowed. Note again that CDS, unlike GDS, does not force you to do things this way. You could theoretically have hundreds of CDS object entries, each of which would contain a different combination of attributes.

3. `Should a directory or an entry be created?`

   When you consider adding information to the namespace, you can choose between creating a new directory, possibly with entries in it, or creating simply one or more entries. When making your decision, take into consideration the following:

   a. Directories cannot be created using XDS; they must be created using administrative commands. Directories are more expensive; they take up more space and take more time to access. However, they can contain entries and can therefore be used to organize information in the namespace.
   b. Entries can be created using XDS and they are cheaper to create and use than directories. However, they must be created in existing directories, and cannot themselves contain other entries.

Access Control for CDS Entries

Each object in the CDS namespace is automatically equipped with a mechanism by which access to it can be regulated by the object’s owner or by another authority. For each object, the mechanism is implemented by a separate list of the entities that can access the object in some way; for example, to read it, write to it, delete it, and so on. Associated with each entity in this list is a string that specifies which operations are allowed for that entity on the object. The object’s list is automatically checked by CDS whenever any kind of access is attempted on that object by any entity. If the entity can be found in the object’s list, and if the kind of access the entity intends is found among its permissions, then the operation is allowed to proceed by CDS; otherwise, it is not allowed.

DCE permission lists are called access control lists (ACLs). ACLs are one of the features of the DCE Security Service used by CDS. ACLs are used to test the entities’ (that is, the principals’) authorization to do things to the objects they propose to do them to. The authorization mechanism for all CDS objects is handled by CDS itself. All that users of the CDS namespace have to do is make sure that ACLs on the CDS objects that they create are set up with the appropriate permissions.

Creation of ACLs

Whenever you create a new entry in the CDS namespace, an ACL is created for it implicitly, and its initial list of entries and their permission sets are determined by the ACL templates associated with the CDS directory in which you create the entry.

Each CDS directory has the following two ACL templates associated with it:

- Initial Container

  This template is used to generate the initial ACL for any directories created within the directory.
- Initial Object

  This template is used to generate ACLs for entries created within the directory.

Like other CDS objects, each CDS directory also has its own ACL, generated from the parent directory's Initial Container template when the child directory is created. The Initial Container template also serves as a template for the child directories' own Initial Container templates.

Manipulating ACLs

There are three ways to manipulate ACLs:
- **acl_edit** command (see the **acl_edit(8sec)** reference pages)
- **dcecp** command (see the **dcecp(8dce)** reference pages)
- DCE ACL application interface (see the **sec_acl_*(3sec)** reference pages)

Initializing ACLs

After creating a CDS directory by using the **cdscp** or **dcecp** command, your first step is usually to run the **acl_edit** command to set up the new directory's ACLs the way you want them. (The new directory will have inherited its ACLs and its templates from the directory in which it was created, as explained in "Creation of ACLs" on page 23.) You may want to modify not only the directory's own ACLs, but also its two templates. To edit the latter, you can specify the **-ic** option (for the Initial Container template) or the **-io** option (for the Initial Object template); otherwise, you will edit the object ACL.

You can modify a directory's ACL templates from an application, assuming that you have control permission for the object, with the same combination of **sec_acl_lookup( )** and **sec_acl_replace( )** calls as for the object ACL. An option to these routines lets you specify which of the three possible ACLs on a directory object you want the call applied to. The ACLs themselves are in identical format.

The **-e** (entry) option to **acl_edit** can be used to make sure that you get the ACL for the specified namespace entry object, and not the ACL (if any) for the object that is *referenced by* the entry. This distinction has to be made clear to **acl_edit** because it finds the object (and hence the ACL) in question by looking it up in the namespace and binding to its ACL manager. Essentially, the **-e** option tells **acl_edit** whether it should bind to the CDS ACL manager (if the entry ACL is wanted), or to the manager responsible for the referenced object's ACL. This latter manager would be a part of the server application whose binding information the entry contained.

An example of such an ambiguous name would be a CDS clearinghouse entry, such as the *cdshostname*_ch entry discussed previously. With the **-e** option, you would edit the ACL on the namespace entry, as follows:

```
acl_edit -e /.:/cdshostname_ch
```

Without the -e option, you would edit the ACL on the clearinghouse itself, which you presumably do *not* want to do.

Similarly, there is a *bind_to_entry* parameter by which the caller of **sec_acl_bind( )** can indicate whether the entry object's ACL or the ACL to which the entry refers is desired. For further details on binding ambiguity, see *IBM DCE Version 3.1 for AIX and Solaris: Application Development Reference*.

Namespace ACLs at Cell Configuration

The ACLs attached to the CDS namespace at configuration are described in *IBM DCE Version 3.1 for AIX and Solaris: Administration Guide*. The following ACL permissions are defined for CDS objects. The single letter in parentheses for each item represents the DCE notation for that permission. These single letters are identical to the untokenized forms returned by **sec_acl_get_printstring( )**.

- read (`r`)

  This permission allows a principal to look up an object entry and view its attribute values.
- write (`w`)

  This permission allows a principal to change an object's modifiable attributes, except for its ACLs.
- insert (`i`)

  This permission allows a principal to create new entries in a CDS directory. It is used with directory entries only.
- delete (`d`)

  This permission allows a principal to delete a name entry from the namespace.
- test (`t`)

  This permission allows a principal to test whether an attribute of an object has a particular value, but does not permit it actually to see any of the attribute values (in other words, read permission for the object is not granted). The test permission allows an application to verify a particular CDS attribute's value without reading it.
- control (`c`)

  This permission allows a principal to modify the entries in the object's ACL. The control permission is automatically granted to the creator of a CDS object.
- administer (`a`)

  This permission allows a principal to issue **cdscp** commands that control the replication of directories. It is used with directory entries only.

Detailed instructions on the mechanics of setting up ACLs on CDS objects can be found in the *IBM DCE Version 3.1 for AIX and Solaris: Administration Guide*.

For CDS directories, read and test permissions are sufficient to allow ordinary principals to access the directory and to read and test entries therein. Principals who you want to be able to add entries in a CDS directory should have insert permission for that directory. Entries created by the RPC NSI routines (for example, when a server exports bindings for the first time) are automatically set up with the correct permissions. However, if you are creating new CDS directories for RPC use, be sure to grant prospective user principals insert permission to the directory so that servers can create entries when they export their bindings. A general list of the permissions required for the various RPC NSI operations as well as the RPC NSI routines (all of whose names are in the form **rpc_ns_...**) are located in the *IBM DCE Version 3.1 for AIX and Solaris: Application Development Reference*.

Note that CDS names do not behave the same way as file system names. A principal does not need to have access to an entire entry name path in order to have access to an entry at the end of that path. For example, a principal can be granted read access to the following entry:

```
/.:/applications/utilities/pr2
```

and yet not have read access to the `utilities` directory itself.

Valid Characters and Naming Rules for CDS

The following subsections discuss the valid character sets for DCE Directory Service names as used by CDS interfaces. They also explain some characters that have special meaning and describe some restrictions and rules regarding case matching, syntax, and size limits.

The use of names in DCE often involves more than one directory service. For example, CDS interacts with either GDS or DNS to find names outside the local cell.

Figure 4 on page 27 details the valid characters in CDS names, and the valid characters in GDS and DNS names as used by CDS interfaces.

Note: Because CDS, GDS, and DNS all have their own valid character sets and syntax rules, the best way to avoid problems is to keep names short and simple, consisting of a minimal set of characters common to all three services. The recommended set is the letters A to Z, a to z, and the digits 0 to 9. In addition to making directory service interoperations easier, use of this subset decreases the probability that users in a heterogeneous hardware and software environment will encounter problems creating and using names.

Although spaces are valid in both CDS and GDS names, a CDS simple name containing a space must be enclosed in ″″ (double quotes) when you enter it through the CDS control program. Additional interface-specific rules are documented in the modules where they apply.

▒ SP	0	▓ @	P	▓ `	p
▓ !	1	A	Q	a	q
▓ "	2	B	R	b	r
▓ #	3	C	S	c	s
▓ $	4	D	T	d	t
▓ %	5	E	U	e	u
▓ &	6	F	V	f	v
▒ '	7	G	W	g	w
▒ (	8	H	X	h	x
▒)	9	I	Y	i	y
▓ *	▒ :	J	Z	j	z
▒ +	▓ ;	K	▓ [	k	▓ {
▒ ,	▓ <	L	▒ \	l	▓ |
-	▒ =	M	▓]	m	▓ }
.	▓ >	N	▓ ^	n	▓ ~
▒ /	▒ ?	O	▓ _	o	

Key: □ **Valid in CDS, GDS, and DNS names**
▒ **Valid only in CDS and GDS names**
▓ **Valid only in CDS names**

Figure 4. Valid Characters in CDS, GDS, and DNS Names

Metacharacters

Certain characters have special meaning to the directory services; these are known as metacharacters. Table 1 on page 28 lists and explains the CDS, GDS, and DNS metacharacters.

Table 1. Metacharacters and Their Meaning

Directory Service	Character	Meaning
CDS	/	Separates elements of a name (simple names).
	*	When used in the rightmost simple name of a name entered in a **cdscp show** or **list** command, acts as a wildcard, matching zero or more characters.
	?	When used in the rightmost simple name of a name entered in a **cdscp show** or **list** command, acts as a wildcard, matching exactly one character.
	\	Used where necessary in front of `*` (asterisk) or `?` (question mark) to escape the character (indicates that the following character is not a metacharacter).
GDS	/	Separates RDNs.
	,	Separates multiple attribute type/value pairs (attribute value assertions) within an RDN.
	=	Separates an attribute type and value in an attribute value assertion.
	\	Used in front of `/` (slash), `,` (comma), or `=` (equal sign) to escape the character (indicates that the following character is not a metacharacter).
DNS	.	Separates elements of a name.

Some metacharacters are not permitted as normal characters within a name. For example, a \ (backslash) cannot be used as anything but an escape character in GDS. You can use other metacharacters as normal characters in a name, provided that you escape them with the backslash metacharacter.

Additional Rules

Table 2 summarizes major points to remember about CDS, GDS, and DNS character sets, metacharacters, restrictions, case-matching rules, internal storage of data, and ordering of elements in a name. For additional details, see the documentation for each technology.

Table 2. Summary of CDS, GDS, and DNS Characteristics

Characteristic	CDS	GDS	DNS
Character Set	a to z, A to Z, 0 to 9 plus space and special characters shown in Figure 4 on page 27	a to z, A to Z, 0 to 9 plus . : , ' + − = ( ) ? / and space	a to z, A to Z, 0 to 9 plus . −
Metacharacters	/ * ? \	/ , = \	.

Table 2. Summary of CDS, GDS, and DNS Characteristics (continued)

Characteristic	CDS	GDS	DNS
Restrictions	Simple names cannot contain slashes. The first simple name following the global cell name (or /.: prefix) cannot contain an equal sign. When entering a name as part of a **cdscp show** or **list** command, you must use a backslash to escape any asterisk or question mark character in the rightmost simple name. Otherwise, the character is interpreted as a wildcard.	Relative distinguished names cannot begin or end with a slash. Attribute types must begin with an alphabetic character, can contain only alphanumerics, and cannot contain spaces. An alternate method of specifying attribute types is by object identifier, a sequence of digits separated by **.** (dots). You must use backslash to escape a slash, a comma, and an equal sign when using them as anything other than metacharacters. Multiple consecutive unescaped occurrences of slashes, commas, equal signs and backslashes are not allowed. Each attribute value assertion contains exactly one unescaped equal sign.	The first character must be alphabetic. The first and last characters cannot be **.** (dot) or – (dash). Cell names in DNS must contain at least one dot; they must be more than one level deep.
Case-Matching Rules	Case exact	Attribute types are matched case insensitive. The case-matching rule for an attribute value can be case exact or case insensitive, depending on the rule defined for its type at the DSA.	Case insensitive
Internal Representation	Case exact	Depends on the case-matching rule defined at DSA. If the rule says case insensitive, alphabetic characters are converted to all lowercase characters. Spaces are removed regardless of the case-matching rule.	Alphabetic characters are converted to all lowercase characters.
Ordering of Name Elements	Big endian (left to right from root to lower-level names).	Big endian (left to right from root to lower-level names).	Little endian (right to left from root to lower-level names).

Maximum Name Sizes

Table 3 lists the maximum sizes for directory service names. Note that the limits are implementation specific, not architectural.

Table 3. Maximum Sizes of Directory Service Names

Name Type	Maximum Bytes
CDS simple name (character string between two slashes)	254
CDS full name (including global or local prefix, cell name, and slashes separating simple names)	1023
GDS relative distinguished name	64
GDS distinguished name	1024
DNS relative name (character string between two dots)	64
DNS fully qualified name (sum of all relative names)	255

Valid Characters for GDS Attributes

This section describes the valid character sets for the GDS attributes.

The values of the country attributes are restricted to the ISO 3166 Alpha-2 code representation of country names. (For more information, see the *IBM DCE Version 3.1 for AIX and Solaris: Administration Guide*.)

The character set for all other naming attributes is the T61 graphical character set. It is described in the next section.

T61 Syntax

The following table shows the T61 graphical character set.

Note: The 1) entry in the table indicates that it is not recommended that you use the codes in Column 2 Row 3, and Column 2 Row 4. Instead, use the appropriate code in Column A.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0			SP	0	@	P		p							Ω	K
1			!	1	A	Q	a	q			¡	±	`		Æ	æ
2			"	2	B	R	b	r			¢	²	´			đ
3			1)	3	C	S	c	s				³	â		ª	
4			1)	4	D	T	d	t			$	×	~		Ħ	ħ
5			%	5	E	U	e	u				µ	¯			
6			&	6	F	V	f	v			#		ˇ		Ĳ	ĳ
7			'	7	G	W	g	w			§				L	l
8			(	8	H	X	h	x				÷	¨			
9			)	9	I	Y	i	y							Ø	ø
A			*	:	J	Z	j	z							Œ	œ
B			+	;	K	[	K				«	»	–		º	ß
C			,	<	L		l	|				¼	—			þ
D			-	=	M	]	m					½	”		Ŧ	ŧ
E			.	>	N		n					¾	¸		η	η
F			/	?	O	_	o					¿	ˇ		'n	

Figure 5. T61 Syntax

The administration interface supports only characters smaller than 0x7e for names. The XDS application programming interface (API) supports the full T61 range as indicated in the preceding table.

Some T61 alphabetical characters have a 2-byte representation. For example, a lowercase letter a with an acute accent is represented by 0xc2 (the code for an acute accent) followed by 0x61 (the code for a lowercase a).

Only certain combinations of diacritical characters and basic letters are valid. They are shown in Figure 6 on page 32.

Name	Repr.	Code	Valid Basic Letters Following
grave accent	`	0xc1	a, A, e, E, i, I, o, O, u, U
acute accent	´	0xc2	a, A, c, C, e, E, g, i, I, l, L, n, N, o, O, r, R, s, S, u, U, y, Y, z, Z
circumflex accent	^	0xc3	a, A, c, C, e, E, g, G, h, H, i, I, j, J, o, O, s, S, u, U, w, W, y, Y
tilde	~	0xc4	a, A, i, I, n, N, o, O, u, U
macron	¯	0xc5	a, A, e, E, i, I, o, O, u, U
breve	˘	0xc6	a, A, g, G, u, U
dot above	˙	0xc7	c, C, e, E, g, G, I, z, Z
umlaut	¨	0xc8	a, A, e, E, i, I, o, O, u, U, y, Y
ring	°	0xca	a, A, u, U
cedilla	¸	0xcb	c, C, G, k, K, l, L, n, N, r, R, s, S, t, T
double accent	˝	0xcd	o, O, u, U
ogonek	˛	0xce	a, A, e, E, i, I, u, U
caron	ˇ	0xcf	c, C, d, D, e, E, l, L, n, N, r, R, s, S, t, T, z, Z

Figure 6. Combinations of Diacritical Characters and Basic Letters

The nonspacing underline (code 0xcc) must be followed by a Latin alphabetical character; that is, a basic letter (a to z or A to Z), or a valid diacritical combination.

Use of OIDs

OIDs are not seen by applications that restrict themselves to using only the RPC NSI routines (**rpc_ns_...( )**), but these identifiers are important for applications that use the XDS interface to read entries directly or to create new attributes for use with namespace entries.

RPC makes use of only four different entry attributes in various application-specified or administrator-specified combinations. CDS, however, contains definitions for many more than these, which can be added by applications to RPC entries through the XDS interface. Attributes that already exist are already properly identified so applications that use these attributes do not have to concern themselves with the OIDs, except to the extent of making sure that they handle them properly.

Unlike UUIDs, OIDs are not generated by command or function call. They originate from ISO, which allocates them in hierarchically organized blocks to recipients. Each recipient, typically some organization, is then responsible for ensuring that the OIDs it receives are used uniquely.

For example, the following OID block was allocated to OSF by ISO:

```
1.3.22
```

OSF can therefore generate, for example, the following OID and allocate it to identify some DCE directory object:

```
1.3.22.1.1.4
```

(The OID `1.3.22.1.1.4` identifies the RPC profile entry object attribute.) OSF is responsible for making sure that `1.3.22.1.1.4` is not used to identify any other attribute. Thus, as long as all OIDs are generated only from within each owner's properly obtained block, and as long as each block owner makes sure that the OIDs generated within its block are properly used, each OID will always be a universally valid identifier for its associated value.

OIDs are encoded and internally represented as strings of hexadecimal digits, and comparisons of OIDs have to be performed as hexadecimal string comparisons (not as comparisons on NULL-terminated strings since OIDs can have NULL bytes as part of their value).

When applications have occasion to handle OIDs, they do so directly because the numbers do not change and should not be reused. However, for users' convenience, CDS also maintains a file (called `cds_attributes`, found in **/opt/dcelocal/etc**) that lists string equivalents for all the OIDs in use in a cell in entries like the following one:

```
1.3.22.1.1.4 RPC_Profile byte
```

This allows users to see `RPC_Profile` in output, rather than the meaningless string `1.3.22.1.1.4`. Further details about the `cds_attributes` file and OIDs can be found in the *IBM DCE Version 3.1 for AIX and Solaris: Administration Guide*.

In summary, the procedure you should follow to create new attributes on CDS entries consists of three steps:

1. Request and receive from your locally designated authority the OIDs for the attributes you intend to create.
2. Update the `cds_attributes` file with the new attributes' OIDs and labels if you want your application to be able to use string name representations for OIDs in output.
3. Using XDS, write the routines to create, add, and access the attributes.

Note: The XDS interface does not look at the **cds_attributes( )** file; therefore, entries cannot be referenced by the string name. You must use the actual OID or define the OID in a header file used by the application.

Your cell administrator should be able to provide you with a name and an OID. The *name* is a guaranteed-unique series of values for a global directory entry name. If the directory is GDS, the name is a series of type/value pairs, such as

```
C=US O=OSF
```

The cell administrator can also obtain an OID block. From this OID space, the administrator can assign you the OIDs you need for your application.

Note that there is no need for new OIDs in connection with cell names. The OIDs for Country Name and Organization Name are part of the X.500 standard implemented in GDS; only the values associated with the OIDs (the values of the objects) change from entry name to entry name. Instead, being able to generate new OIDs gives you the ability to invent and add new details to the directory itself. For example, you can create new kinds of CDS entry attributes by generating new OIDs to identify them. The same thing can be done to GDS, although the procedure is more complicated because it involves altering the directory schema.

Chapter 3. XDS and the DCE Cell Namespace

This chapter describes the use of the XDS programming interface when accessing the CDS namespace. The first section provides an introduction to using XDS in the CDS namespace. XDS objects describe XDS objects and how they are used to access CDS data. Accessing CDS using XDS provides a step-by-step procedure for writing an XDS program to access CDS. Object—handling techniques provides examples of using the XOM interface to manipulate objects. XDS/CDS object recipes provides details of the structure of XDS/CDS objects. Finally, attribute and data type translation, provides translation tables between XDS and CDS for attributes and data types.

Introduction to Accessing CDS with XDS

Outside of the DCE cells and their separate namespaces is the global namespace in which the cell names themselves are entered, and where all intercell references are resolved. Two directory services participate in the global namespace. The first is the X.500-compliant GDS. The second is DNS, with which DCE interacts, but it is not a part of DCE.

The global and cell directory services are accessed implicitly by RPC applications using the NSI interface. GDS and CDS can also be accessed explicitly by using the XDS interface. With XDS, application programmers can gain access to GDS, a powerful general-purpose distributed database service, which can be used for many other things besides intercell binding. XDS can also be used to access the *cell* namespace directly, as this chapter describes.

An XDS application looks very different from the RPC-based DCE applications. This is partly because there is no dependency in XDS on the DCE RPC interface, although you can use both interfaces in the same application. Also, XDS is a generalized directory interface, oriented more toward performing large database operations than toward fine-tuning the contents of RPC entries. Nevertheless, XDS can be used as a general access mechanism on the CDS namespace.

Using the Reference Material in This Chapter

Complete descriptions of all the XDS and XOM functions used in CDS operations can be found in the *IBM DCE Version 3.1 for AIX and Solaris: Application Development Reference*, which you should have beside you as you read through the examples in this chapter. In particular, refer to that manual for information about XDS error objects, which are not discussed in this chapter.

Complete descriptions for all objects required as *input* parameters by XDS functions when accessing a CDS namespace can be found in “XDS/CDS Object Recipes” on page 63. Abbreviated definitions for these same objects can be found with all the others in “Part 4. XDS/XOM Supplementary Information” on page 179. XOM functions are general-purpose utility routines that operate on objects of any class, and take the rest of their input in conventional form.

Slightly less detailed descriptions of the *output* objects you can expect to receive when accessing CDS through XDS are also given in “XDS/CDS Object Recipes” on page 63. You do not have to construct objects of these classes yourself; you just have to know their general structure so that you can disassemble them using XOM routines.

No information is given in this chapter about the `DS_status` error objects that are returned by unsuccessful XDS functions; a description of all the subclasses of `DS_status` requires a chapter in itself. Code for a rudimentary general-purpose **DS_status** handling routine is located in the **cds_xmpl.c** XDS sample program in the **/opt/dcelocal/examples/xdsxon** directory.

What You Cannot Do with XDS

XDS allows you to perform general operations on CDS entry attributes, something which you cannot do through the DCE RPC NSI interface. However, there are certain things you cannot do to cell directory entries even through XDS:

- You cannot create or modify directory entries; the **ds_modify_rdn( )** function does not work in a CDS namespace. These operations must be performed through the CDS control program (**cdscp**) or DCE control program (**dcecp**). For more information, see the *IBM DCE Version 3.1 for AIX and Solaris: Administration Commands Reference*.
- You cannot perform XDS searches on the cell namespace; the XDS function **ds_search( )** does not work. This is mainly because the CDS has no concept of a hierarchy of entry attributes, such as the X.500 schema. The **ds_compare( )** function, however, does work.
- You cannot modify ACL entries in the cell namespace.

Registering A Nonlocal Cell

If you are planning to use XDS to access the cell namespace in a one-cell environment (that is, your cell does not need to communicate with other DCE cells), you do not need to set up a cell entry in GDS for your cell because the XDS functions simply call the appropriate statically linked CDS routines to access the cell namespace. XDS, in conjunction with CDS, is able to recognize the local cell portion of a name in the cell namespace without help from GDS or DNS. You need to specify the complete name for the interface object *DS_C_DS_DN*; omitting the cell name portion of the name is not allowed. See a description of the **DS_C_DS_DN** object in “Building a Name Object” on page 42.

For XDS to be able to access any nonlocal cell namespace, that cell must be registered (that is, have an entry) in the global namespace. GDS is not provided in DCE AIX version 4.1 and greater.

For information on setting up your cell name, see the *IBM DCE Version 3.1 for AIX and Solaris: Administration Guide*.

XDS Objects

The XDS interface differs from the other DCE component interfaces in that it is *object oriented*. The following subsections explain two things: first, what object-oriented programming means in terms of using XDS; and second, how to use XDS to access CDS.

Imagine a generalized data structure that always has the same data *type*, and yet can contain any kind of data, and any amount of it. Functions could pass these structures back and forth in the same way all the time, and yet they could use the same structures for any kind of data they wanted to store or transfer. Such a data structure, if it existed, would be a true *object*. Programming language constructs allow interfaces to pretend that they use objects, although the realities of implementation are not usually so simple.

XDS is such an interface. For the most part, XDS functions neither accept nor return values in any form but as objects. The objects themselves are indeed always the same data type; namely, pointers to arrays of *object descriptor* (C `struct`) elements. Contained within these `OM_descriptor` element structures are unions that can actually accommodate all the different kinds of values an object can be called on to hold. In order to allow the interface to make sense of the unions, each `OM_descriptor` also contains a `syntax` field, which indicates the data type of that descriptor's union. There is also a record of what the descriptor's value actually is; for example, whether it is a name, a number, an address, a list, and so on. This information is held in the descriptor's `type` field.

These `OM_descriptor` elements, which are referred to as *object descriptors* or *descriptors*, are the basic building blocks of *all* XDS objects; every actual XDS object reduces to arrays of them. Each descriptor contains three items:

- A `type` field, which identifies the descriptor's value
- A `syntax` field, which indicates the data type of the `value` field
- The `value` field, which is a union

Figure 7 illustrates one such object descriptor.

type:OM_CLASS
syntax:OID string
value:DS_C_DS_DN

Figure 7. One Object Descriptor

Note that, from an abstract point of view, `syntax` is just an implementation detail. The scheme really consists only of a type/value pair. The `type` gives an identity to the object (something like CDS entry attribute, CDS entry name, or DUA access point), and the `value` is some data associated with that identity, just as a variable has a name that gives meaning to the value it holds, and the value itself.

In order to make the representation of objects as logical and as flexible as possible, these two logical components of every object, `type` and `value`, are themselves each represented by separate object descriptors. Thus, the first element of every complete object descriptor array is a descriptor whose `type` field identifies its `value` field as containing the name of the kind (or *class*) of this object, and the `syntax` field indicates how that name `value` should be read. Next is usually one (or more, if the object is multivalued) object descriptor whose `type` field identifies its `value` field as containing some value appropriate for this class of object. Finally, every complete object descriptor array ends with a descriptor element that is identified by its fields as being a NULL-terminating element.

Thus, a minimum-size descriptor array consists of just two elements: the first contains its class identity, and the second is a NULL (it is legitimate for objects not to have values). When an object does have a value, it is held in the `value` field of a descriptor whose `type` field communicates the value's meaning.

Figure 8 illustrates the arrangement of a complete object descriptor array.

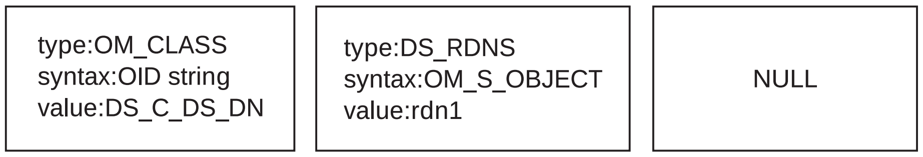


Figure 8. A Complete Object Represented

Object Attributes

The generic term for any object value is *attribute*. In this sense, an object is nothing but a collection of attributes, and every object descriptor describes one attribute. The first attribute's value identifies the object's class, and this determines all the other attributes the object is supposed to have. One or more other attributes follow, which contain the object's working values. The NULL object descriptor at the end is an implementation detail, and is not a part of the object.

Note that, depending on the attribute it represents, a descriptor's `value` field can contain a pointer to another array of object descriptors. In other words, an object's value can be another object.

Figure 9 shows a three-layer compound object: the top-level superobject, `dn_object`, contains the subobject `rdn1`, which in turn contains the subobject `ava1`.

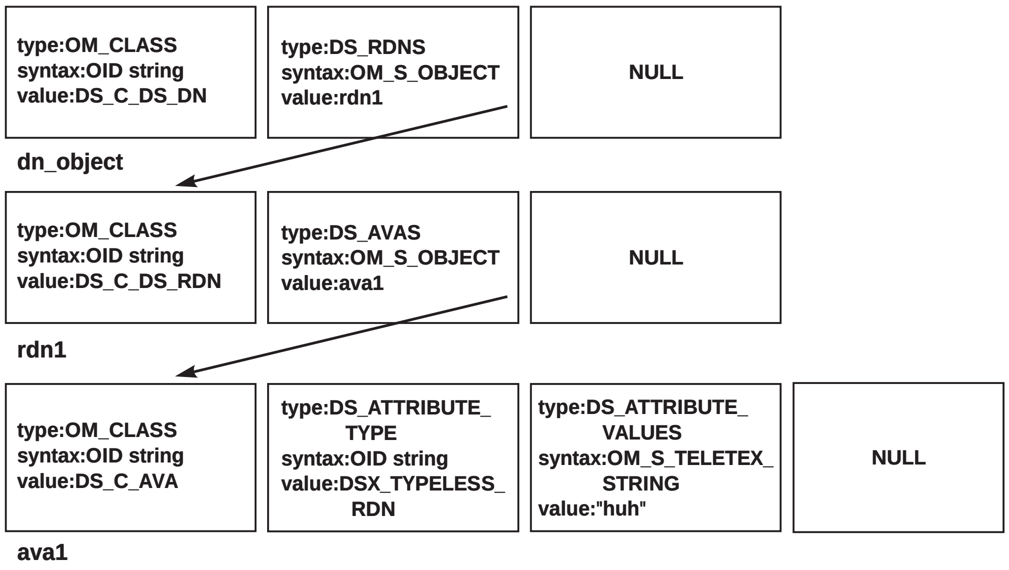


Figure 9. A Three-Layer Compound Object

Interface Objects and Directory Objects

GDS is composed of directory objects that reflect the X.500 design. The XDS interface also works with objects. However, there is a big difference between directory and XDS objects. Programmers do not work directly with the directory objects; they are composed of attributes that make up the directory service's implementation of entries.

Programmers work with XDS objects. XDS objects have explicit data representations that can be directly manipulated with programming language operators. Some of these techniques are described in this chapter; are located in the sample files located in the **/opt/dcelocal/examples/xdsxom** directory.

XDS and GDS terminology sometimes suggests that XDS objects are somehow direct representations of the directory objects to which they communicate information. This is not the case, however. You never directly see or manipulate the directory objects; the XDS interface objects are used only to pass parameters to the XDS calls, which in turn request GDS (or CDS) to perform operations on the directory objects. The XDS objects are therefore somewhat arbitrary structures defined by the interface.

Figure 10 on page 40 illustrates the relationship between XDS (also called *interface*) objects and directory objects. The figure shows an application passing several properly initialized XDS objects to some XDS function; it then takes some action, which affects the attribute contents of certain directory objects. The application never works with the directory objects; it works with the XDS interface objects.

A side effect of the existence of a separate XDS interface and GDS or CDS directory objects is the existence of attributes for both kinds of objects as well. Because the purpose of XDS objects is to feed data into and extract data from directory objects, programmers work with XDS objects whose attributes have *directory* object attributes as their values. You should keep in mind the distinction between directory objects and interface objects.

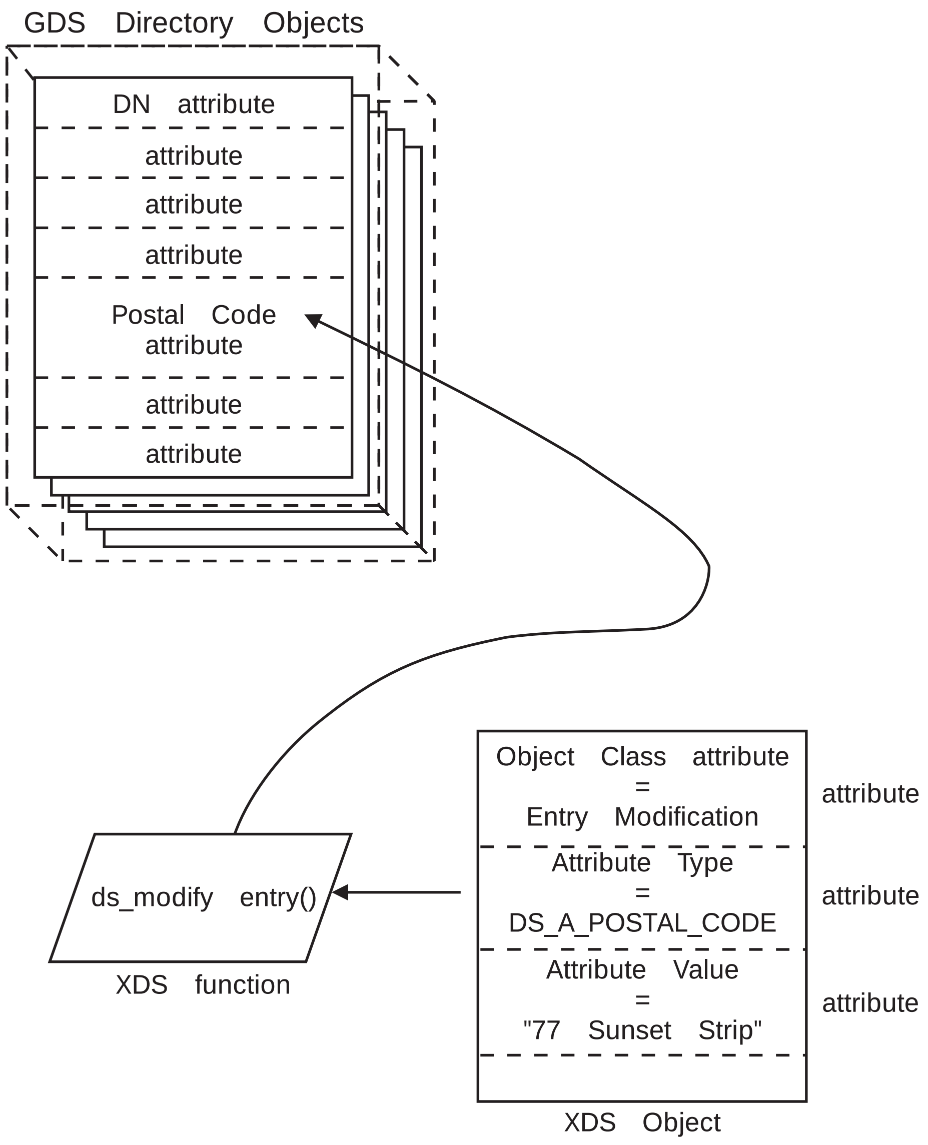


Figure 10. Directory Objects and XDS Interface Objects

Directory Objects and Namespace Entries

The GDS namespace is a hierarchical collection of entries. The name of each of these entries is an attribute of a directory object. The object is accessed through XDS by stating its name attribute.

Figure 11 on page 41 shows the relationship of entry names in the GDS namespace to the GDS directory objects to which they refer.

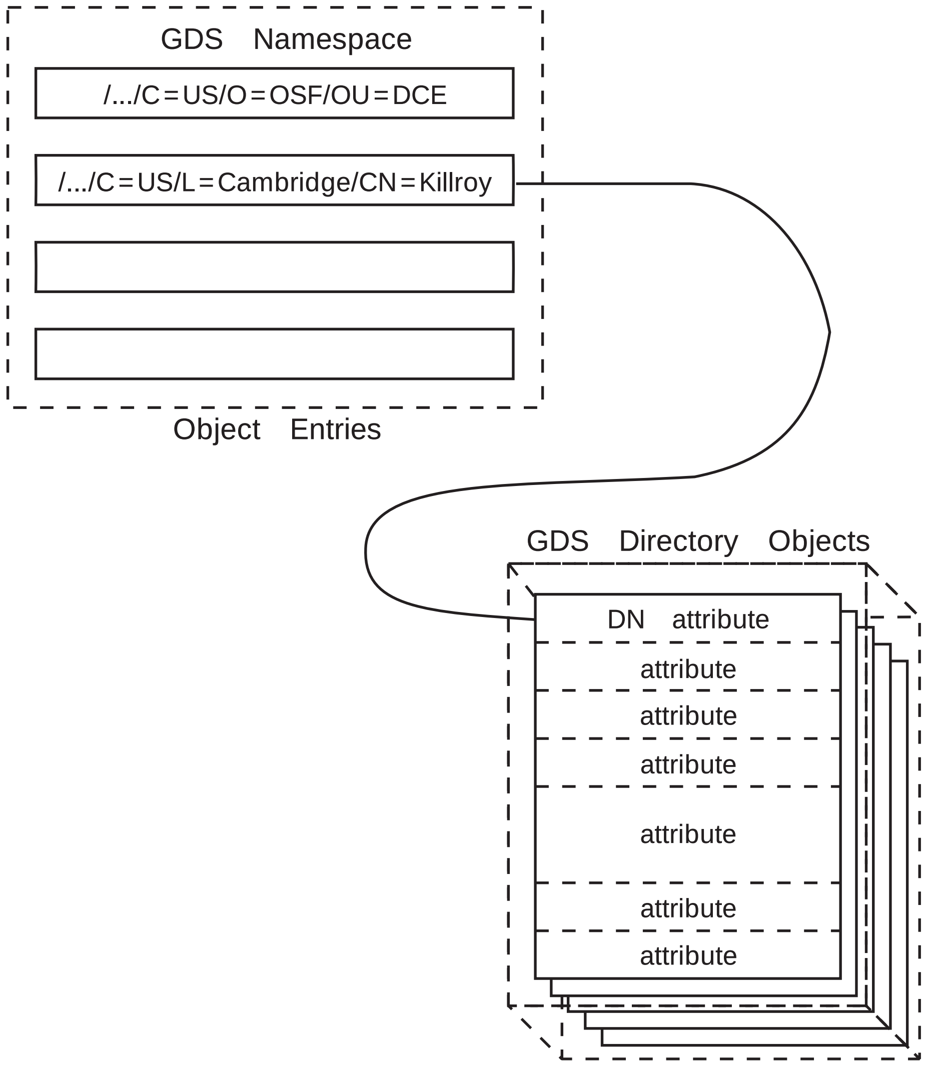


Figure 11. Directory Objects and Namespace Entries

Values That an Object Can Contain

There are many different classes of objects defined for the XDS interface; still more are defined by the X.500 standard for general directory use. But only a small number of classes are needed for XDS/CDS operations, and only those classes are discussed in this chapter. Information about other classes can be found in Part 4 of this guide.

The class that an object belongs to determines what sort of information the object can contain. Each object class consists of a list of attributes that objects must have. For example, you would expect an object in the directory entry name class to be required to have an attribute to hold the entry name string. However, it is not sufficient to simply place a string like the following into an object descriptor:

```
/.../C=US/O=OSF/OU=DCE/hosts/tamburlaine/self
```

A full directory entry name such as the preceding one is called in XDS a distinguished name (DN), meaning that the entry name is fully qualified (distinct) from root to entry name. To properly represent the entry name in an object, you must look up the definition of the XDS distinguished name object class and build an object that has the set of attributes that the definition prescribes.

Building a Name Object

Complete definitions for all the object classes required as input for XDS functions can be found in “XDS/CDS Object Recipes” on page 63. Among them is the class for distinguished name objects, called `DS_C_DS_DN`. There you will learn that this class of object has two attributes: its class attribute, which identifies it as a `DS_C_DS_DN` object, and a second attribute, which occurs multiple times in the object. Each instance of this attribute contains as its value one piece of the full name; for example, the directory name `hosts`.

The `DS_C_DS_DN` attribute that holds the entry name piece, or relative distinguished name (RDN), is defined by the class rules to hold, not a string, but another object of the RDN class (`DS_C_DS_RDN`).

Thus, a static declaration of the descriptor array representing the `DS_C_DS_DN` object would look like the following:

```
static OM_descriptor    Full_Entry_Name_Object[] = {

     OM_OID_DESC(OM_CLASS, DS_C_DS_DN),
/*  ^^^^^^^^^^^*/
/*   Macro to put an "OID string" in a descrip-         */
/*     tor's type field and fill its other              */
/*     fields with appropriate values.                  */

     {DS_RDNS, OM_S_OBJECT, {0, Country_RDN}},
/*     ^^^^^^^    ^^^^^^^^^^^    ^^^^^^^^^^^ */
/*     type      syntax            value                */
/*                                                      */
/*     (the "value" union is in fact here a             */
/*      structure; the 0 fills a pad field in           */
/*      that structure.)                                */

     {DS_RDNS, OM_S_OBJECT, {0, Organization_RDN}},
     {DS_RDNS, OM_S_OBJECT, {0, Org_Unit_RDN}},
     {DS_RDNS, OM_S_OBJECT, {0, Hosts_Dir_RDN}},
     {DS_RDNS, OM_S_OBJECT, {0, Tamburlaine_Dir_RDN}},
     {DS_RDNS, OM_S_OBJECT, {0, Self_Entry_RDN}},

     OM_NULL_DESCRIPTOR
/*
^^^^^^^^^^^^^^^^^^
                                                   */
/*    Macro to fill a descriptor with proper            */
/*      NULL values.                                    */

};
```

The use of the `OM_OID_DESC` and `OM_NULL_DESCRIPTOR` macros slightly obscures the layout of this declaration. However, each line contains code to initialize exactly one `OM_descriptor` object; the array consists of eight objects.

The names (such as `Country_RDN`) in the descriptors’ `value` fields refer to the other descriptor arrays, which separately represent the relative name objects. (The order of the C declaration in the source file is opposite to the order described here.) Because `DS_C_DS_RDN` objects are now called for, the next step is to look at what attributes that class requires.

The definition for DS_C_DS_RDN can be found in “The DS_C_DS_RDN Object” on page 69 . This class object is defined, like DS_C_DS_DN, to have only one attribute (with the exception of the OM_Object attribute, which is mandatory for all objects). The one attribute, DS_AVAS, holds the value of one relative name. The syntax of this value is OM_S_OBJECT, meaning that DS_AVAS’s value is a pointer to yet another object descriptor array:

```
static OM_descriptor    Country_RDN[] = {

     OM_OID_DESC(OM_CLASS, DS_C_DS_RDN),

     {DS_AVAS, OM_S_OBJECT, {0, Country_Value}},

     OM_NULL_DESCRIPTOR
};
```

Note that there should also be five other similar declarations, one for each of the other DS_C_DS_RDN objects held in DS_C_DS_DN.

The declarations have the same meanings as they did in the previous example. Country_Value is the name of the descriptor array that represents the object of class DS_C_AVA, which we are now about to look up.

The rules for the DS_C_AVA class can be found in this chapter just after DS_C_DS_RDN. They tell us that DS_C_AVA objects have two attributes aside from the omnipresent OM_Object; namely:

- DS_ATTRIBUTE_VALUES

  This attribute holds the object’s value.
- DS_ATTRIBUTE_TYPE

  This attribute gives the meaning of the object’s value.

In this instance, the meaning of the string US is that it is a country name. There is a particular directory service attribute value for this; it is identified by an OID that is associated with the label DS_A_COUNTRY_NAME (the OIDs held in objects are represented in string form). Accordingly, we make that OID the value of DS_ATTRIBUTE_TYPE, and we make the name string itself the value of DS_ATTRIBUTE_VALUES, as shown.

```
static OM_descriptor    Country_Value[]= {

    OM_OID_DESC(OM_CLASS, DS_C_AVA),

    OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_COUNTRY_NAME),

    {DS_ATTRIBUTE_VALUES, OM_S_PRINTABLE_STRING, OM_STRING("US")},
/*
^^^^^^^^^^^^^^^ */
/*                                          Macro to properly */
/*       fill the "value" union with the NULL-terminated string. */

     OM_NULL_DESCRIPTOR
};
```

There are also five other DS_C_AVA declarations, one for each of the five other separate name piece objects referred to in the DS_C_DS_RDN superobjects.

A Complete Object

The previous sections described how an object is created: you look up the rules for the object class you require, and then add the attributes called for in the definition. Whenever some attribute is defined to have an object as its value, you have to look up the class rules for the new object and declare a further descriptor array for it. In this way, you continue working down through layers of subobjects until you reach an object class that contains no subobjects as values; at that point, you are finished.

Normally, you do not statically declare objects in real applications. The steps outlined in the preceding text are given as a method for determining what an object looks like. Once you have done that, you can then write routines to create the objects dynamically. An example of how to do this is located in the **teldir.c** example application in the sample files located in the **/opt/dcelocal/examples/xdsxom** directory.

The process of object building is somewhat easier than it sounds. There are only five different object classes needed for input to XDS functions when accessing CDS, and only one of those, the `DS_C_DS_DN` class, has more than one level of subobjects. The rules for all five of these classes can be found in "Chapter 5. XOM Programming" on page 83 of this guide. In order to use the GDS references, you should know a few things about class hierarchy.

Class Hierarchy

Object classes are hierarchically organized so that some classes may be located above some classes in the hierarchy and below others in the hierarchy. In any such system of subordinate classes, each next lower class inherits all the attributes prescribed for the class immediately above it, plus whatever attributes are defined peculiarly for it alone. If the hierarchy continues further down, cumulative collection of attributes continues to accumulate. If there were a class for every letter of the alphabet, starting at the highest level with A and continuing down to the lowest level with Z, and if each succeeding letter was a subclass of its predecessor, the Z class would possess all the attributes of all the other letters, as well as its own, while the A class would possess only the A class attributes.

XDS/XOM classes are seldom nested more than two or at most three layers. All inherited attributes are explicitly listed in the object descriptions that follow, so you do not have to worry about class hierarchies here. However, the complete descriptions of XDS/XOM objects in "Part 4. XDS/XOM Supplementary Information" on page 179 of this guide rely on statements of class inheritance to fill out their attribute lists for the different classes. Refer to "Part 4. XDS/XOM Supplementary Information" on page 179 for information about the classes of objects that can be returned by XDS calls in order to be able to handle those returned objects.

Class Hierarchy and Object Structure

Note that *class hierarchy* is different from *object structure*. Object structure is the layering of object arrays that was previously described in the `DS_C_DS_DN` declaration in "Building a Name Object" on page 42. It occurs when one object contains another object as the value of one or more of its attributes.

This is what is meant by recursive objects: one object can point to another object as one of its attribute values. The layering of subobjects below superobjects in this way is described repeatedly in “XDS/CDS Object Recipes” on page 63.

The only practical significance of class hierarchy is in determining all the attributes a certain object class must have. Once you have done this, you may find that a certain attribute requires as its value some other object. The result is a compound object, but this is completely determined by the attributes for the particular class you are looking at.

Public and Private Objects and XOM

In “Building a Name Object” on page 42, you saw how a multilevel XDS object can be statically declared in C code. Now imagine that you have written an application that contains such a static `DS_C_DS_DN` object declaration. From the point of view of your application, that object is nothing but a series of arrays, and you can manipulate them with all the normal programming operators, just as you can any other data type. Nevertheless, the object is syntactically perfectly acceptable to any XDS (or XOM) function that is prepared to receive a `DS_C_DS_DN` object.

Objects are also created by the XDS functions themselves; this is the way they usually return information to callers. However, there is a difference between objects generated by the XDS interface and objects that are explicitly declared by the application: you cannot access the former, *private*, objects in the direct way that you can the latter, *public*, objects.

These two kinds of objects are the same as far as their classes and attributes are concerned. The only difference between them is in the way they are accessed. The public objects that an application explicitly creates or declares in its own memory area are just as accessible as any of the other data storage it uses. However, private objects are created and held in the XDS interface’s own system memory. Applications get handles to private objects, and, in order to access the private objects’ contents, they have to pass the handles to object management functions. The object management (XOM) functions make up a sort of all-purpose companion interface to XDS. Whereas XDS functions typically require some specific class object as input, the XOM functions accept objects of any class and perform useful operations on them.

If a private object needs to be manipulated, one of the XOM functions, **om_get( )**, can be called to make a public copy of the private object. Then, calling the XOM **om_create( )** function allows applications to generate private objects manipulable by **om_get( )**. The main significance of private as opposed to public objects is that they do not have to be explicitly operated on; instead, you can access them cleanly through the XOM interface and let it do most of the work. You still have to know something about the objects’ logical representation, however, to use XOM.

Except for a few more details, which will be mentioned as needed, this is practically all there is to XDS object representation.

XOM Objects and XDS Library Functions

To call an XDS library function, do the following:

1. Decide what input parameters you must supply to the function.

2. Bundle up these parameters in objects (that is, arrays of object descriptors) in an XDS format.

Almost all data returned to you by an XDS function is enclosed in objects, which you must parse to recover the information that you want. This task is made almost automatic by a library function supplied with the companion X/Open OSI-Abstract-Data Manipulation (XOM) interface.

With XDS, the programmer has to perform a lot of call parameter management, but in other respects the interface is easy to use. The XDS functions' dependence on objects makes them easy to call, once you have the objects themselves correctly set up.

Accessing CDS Using the XDS Step-by-Step Procedure

You now know all that you need to know to work with a cell namespace through XDS. The following subsections provide a walk-through of the steps of some typical XDS/CDS operations. They describe what is involved in using XDS to access existing CDS attributes. They then describe how you can create and access new CDS entry attributes.

Reading and Writing Existing CDS Entry Attributes With XDS

Suppose that you want to use XDS to read some information from the following CDS entry:

`/.../C=US/O=OSF/OU=DCE/hosts/tamburlaine/self`

As explained in the *IBM DCE Version 3.1 for AIX and Solaris: Administration Guide*, the **/.:/hosts/** *hostname/*`self` entry, which is created at the time of cell configuration, contains binding information for the machine *hostname*. Since this is a simple RPC NSI entry, there is not very much in the entry that is interesting to read, but this entry is used as an example anyway as a simple demonstration.

Following are the header inclusions and general data declarations:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <xom.h>
#include <xds.h>
#include <xdsbdcp.h>
#include <xdscds.h>
```

Note that the **xom.h** and **xds.h** header files must be included in the order shown in the preceding example. Also note that the **xdscds.h** header file is brought in for the sake of `DSX_TYPELESS_RDN`. This file is where the CDS-significant OIDs are defined. The **xdsbdcp.h** file contains information necessary to the Basic Directory Contents Package, which is the basic version of the XDS interface you can use in this program.

The XDS/XOM interface defines numerous object identifier string constants, which are used to identify the many object classes, parts, and pieces (among other things) that it needs to know about. In order to make sure that these OID constants do not collide with any other constants, the interface refers to them with the string `OMP_O_` prefixed to the user-visible form; for example, `DS_C_DS_DN` becomes

OMP_O_DS_C_DS_DN internally. In order to make application instances consistent with the internal form, use OM_EXPORT to import *all* XDS-defined or XOM-defined OID constants used in your application.

```
OM_EXPORT( DS_A_COUNTRY_NAME )
OM_EXPORT( DS_A_OBJECT_CLASS )
OM_EXPORT( DS_A_ORG_UNIT_NAME )
OM_EXPORT( DS_A_ORG_NAME )

OM_EXPORT( DS_C_ATTRIBUTE )
OM_EXPORT( DS_C_ATTRIBUTE_LIST )
OM_EXPORT( DS_C_AVA )
OM_EXPORT( DS_C_DS_DN )
OM_EXPORT( DS_C_DS_RDN )
OM_EXPORT( DS_C_ENTRY_INFO_SELECTION )
OM_EXPORT( DSX_TYPELESS_RDN )

/* ... Special OID for an untyped (that is, nonX.500)       */
/* relative distinguished name. Defined in xdscds.h header.
*/
```

A further important effect of OM_EXPORT is that it builds an OM_string structure to hold the exported OID hexadecimal string. As explained in the previous chapter, OIDs are not numeric values, but strings. Comparisons and similar operations on OIDs must access them as strings. Once an OID has been exported, you can access it by using its declared name. For example, the hexadecimal string representation of DS_C_ATTRIBUTE is contained in DS_C_ATTRIBUTE.elements, and the length of this string is contained in DS_C_ATTRIBUTE.length.

Significance of Typed and Untyped Entry Names

Next are the static declarations for the lowest layer of objects that make up the global name (distinguished name) of the CDS directory entry you want to read. These lowest-level objects contain the string values for each part of the name. Remember that the first three parts of the name (excluding the global prefix **/.../**, which is not represented) constitute the cell name, as follows:

```
/C=US/O=OSF/OU=DCE/
```

In this example, assume that GDS is being used as the cell’s global directory service, so the cell name is represented in X.500 format, and each part of it is typed in the object representation; for example, DS_A_COUNTRY_NAME is the DS_ATTRIBUTE_TYPE in the Country_String_Object. If you were using DNS, the cell name might be something like the following:

```
osf.org.dce
```

In this case, the entire string osf.org.dce would be held in a single object whose DS_ATTRIBUTE_TYPE would be DSX_TYPELESS_RDN.

DSX_TYPELESS_RDN is a special type that marks a name piece as not residing in an X.500 namespace. If the object resides under a typed X.500 name, as is the case in the declared object structures, then it serves as a delimiter for the end of the cell name GDS looks up, and the beginning of the name that is passed to a CDS server in that cell, assuming that the cell has access to GDS; if not, such a name cannot be resolved. In the following name, the untyped portion is at the beginning:

```
/.../osf.org.dce/hosts/zenocrate/self
```

In this case, the name is passed immediately by XDS via the local CDS (and the GDA) to DNS for resolution of the cell name. Thus, the typing of entry names determines which directory service a global directory entry name is sent to for resolution.

Static Declarations

The following are the static declarations you need:

```
/****************************************************************/
/* Here are the objects that contain the string values for each  */
/*  part of the CDS entry's global name...                       */

static OM_descriptor    Country_String_Object[] = {
 OM_OID_DESC(OM_CLASS, DS_C_AVA),
 OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_COUNTRY_NAME),
 {DS_ATTRIBUTE_VALUES, OM_S_PRINTABLE_STRING, OM_STRING("US")},
 OM_NULL_DESCRIPTOR
};

static OM_descriptor    Organization_String_Object[] = {
 OM_OID_DESC(OM_CLASS, DS_C_AVA),
 OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_ORG_NAME),
 {DS_ATTRIBUTE_VALUES, OM_S_PRINTABLE_STRING, OM_STRING("OSF")},
 OM_NULL_DESCRIPTOR
};

static OM_descriptor    Org_Unit_String_Object[] = {
 OM_OID_DESC(OM_CLASS, DS_C_AVA),
 OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_ORG_UNIT_NAME),
 {DS_ATTRIBUTE_VALUES, OM_S_PRINTABLE_STRING, OM_STRING("DCE")},
 OM_NULL_DESCRIPTOR
};

static OM_descriptor    Hosts_Dir_String_Object[] = {
 OM_OID_DESC(OM_CLASS, DS_C_AVA),
 OM_OID_DESC(DS_ATTRIBUTE_TYPE, DSX_TYPELESS_RDN),
 {DS_ATTRIBUTE_VALUES, OM_S_PRINTABLE_STRING, OM_STRING("hosts")},
 OM_NULL_DESCRIPTOR
};

static OM_descriptor    Tamburlaine_Dir_String_Object[] = {
 OM_OID_DESC(OM_CLASS, DS_C_AVA),
 OM_OID_DESC(DS_ATTRIBUTE_TYPE, DSX_TYPELESS_RDN),
 {DS_ATTRIBUTE_VALUES, OM_S_PRINTABLE_STRING, OM_STRING("tamburlaine")},
 OM_NULL_DESCRIPTOR
};
static OM_descriptor    Self_Entry_String_Object[] = {
 OM_OID_DESC(OM_CLASS, DS_C_AVA),
 OM_OID_DESC(DS_ATTRIBUTE_TYPE, DSX_TYPELESS_RDN),
 {DS_ATTRIBUTE_VALUES, OM_S_PRINTABLE_STRING, OM_STRING("self")},
 OM_NULL_DESCRIPTOR
};
```

The string objects are contained by a next-higher level of objects that identify the strings as being pieces (RDNs) of a fully qualified directory entry name (DN). Thus, the `Country_RDN` object contains `Country_String_Object` as the value of its `DS_AVAS` attribute; `Organization_RDN` contains `Organization_String_Object`, and so on.

```
/****************************************************************/
/* Here are the "relative distinguished name" objects.

static OM_descriptor    Country_RDN[] = {
 OM_OID_DESC(OM_CLASS, DS_C_DS_RDN),
 {DS_AVAS, OM_S_OBJECT, {0, Country_String_Object}},
 OM_NULL_DESCRIPTOR
};

static OM_descriptor    Organization_RDN[] = {
 OM_OID_DESC(OM_CLASS, DS_C_DS_RDN),
 {DS_AVAS, OM_S_OBJECT, {0, Organization_String_Object}},
 OM_NULL_DESCRIPTOR
};

static OM_descriptor    Org_Unit_RDN[] = {
 OM_OID_DESC(OM_CLASS, DS_C_DS_RDN),
 {DS_AVAS, OM_S_OBJECT, {0, Org_Unit_String_Object}},
 OM_NULL_DESCRIPTOR
};

static OM_descriptor    Hosts_Dir_RDN[] = {
 OM_OID_DESC(OM_CLASS, DS_C_DS_RDN),
 {DS_AVAS, OM_S_OBJECT, {0, Hosts_Dir_String_Object}},
 OM_NULL_DESCRIPTOR
};

static OM_descriptor    Tamburlaine_Dir_RDN[] = {
 OM_OID_DESC(OM_CLASS, DS_C_DS_RDN),
 {DS_AVAS, OM_S_OBJECT, {0, Tamburlaine_Dir_String_Object}},
 OM_NULL_DESCRIPTOR
};
static OM_descriptor    Self_Entry_RDN[] = {
 OM_OID_DESC(OM_CLASS, DS_C_DS_RDN),
 {DS_AVAS, OM_S_OBJECT, {0, Self_Entry_String_Object}},
 OM_NULL_DESCRIPTOR
};
```

At the highest level, all the subobjects are gathered together in the DN object named `Full_Entry_Name_Object`.

```
/***********************************************************/

static OM_descriptor    Full_Entry_Name_Object[] = {
     OM_OID_DESC(OM_CLASS, DS_C_DS_DN),
     {DS_RDNS, OM_S_OBJECT, {0, Country_RDN}},
     {DS_RDNS, OM_S_OBJECT, {0, Organization_RDN}},
     {DS_RDNS, OM_S_OBJECT, {0, Org_Unit_RDN}},
     {DS_RDNS, OM_S_OBJECT, {0, Hosts_Dir_RDN}},
     {DS_RDNS, OM_S_OBJECT, {0, Tamburlaine_Dir_RDN}},
     {DS_RDNS, OM_S_OBJECT, {0, Self_Entry_RDN}},
     OM_NULL_DESCRIPTOR
};
```

Other Necessary Objects for ds_read( )

The **ds_read( )** procedure takes requests in the form of a `DS_C_ENTRY_INFO_SELECTION` class object. However, if you refer to the recipe for this object class in “XDS/CDS Object Recipes” on page 63, you will find that it is much simpler than the name object; it contains no subobjects, and its declaration is straightforward.

The value of the `DS_ALL_ATTRIBUTES` attribute specifies that all attributes be read from the CDS entry, which is specified in the `Full_Entry_Name_Object` variable.

Note that the term *attribute* is used slightly differently in CDS and XDS contexts. In XDS, attributes describe the values that can be held by various object classes; they can be thought of as object fields. In CDS, attributes describe the values that can be associated with a directory entry. The following code fragment shows the definition of a DS_C_ENTRY_INFO_SELECTION object:

```
static OM_descriptor    Entry_Info_Select_Object[] = {

    OM_OID_DESC(OM_CLASS, DS_C_ENTRY_INFO_SELECTION),
    {DS_ALL_ATTRIBUTES, OM_S_BOOLEAN, OM_TRUE},
    {DS_INFO_TYPE, OM_S_ENUMERATION, DS_TYPES_AND_VALUES},
    OM_NULL_DESCRIPTOR
};
```

Miscellaneous Declarations

The following are declarations for miscellaneous variables:

```
OM_workspace          xdsWorkspace;
     /* ...will contain handle to our "workspace"  */

DS_feature featureList[] = {

     { OM_STRING(OMP_O_DS_BASIC_DIR_CONTENTS_PKG), OM_TRUE },
     { 0 }
};
     /* ...list of service "packages" we will want from XDS  */

OM_private_object     session;
     /* ...will contain handle to a bound-to directory session  */


DS_status             dsStatus;
     /* ...status return from XDS calls  */

OM_return_code        omStatus;
     /* ...status return from XOM calls  */

OM_sint               dummy;
     /* ...for unsupported ds_read() argument  */

OM_private_object     readResultObject;
     /* ...to receive entry information read from CDS by "ds_read()" */
OM_type I_want_entry_object[] = {DS_ENTRY, OM_NO_MORE_TYPES};
OM_type I_want_attribute_list[] = {DS_ATTRIBUTES, OM_NO_MORE_TYPES};
OM_type I_want_attribute_value[] = {DS_ATTRIBUTE_VALUES, \
                                    OM_NO_MORE_TYPES};
     /* ...arrays to pass to "om_get()" to extract subobjects */
     /*  from the result object returned by "ds_read()"       */
OM_value_position number_of_descriptors;
     /* ...to hold number of attribute descriptors returned   */
     /*  by "om_get()                                         */

OM_public_object entry;
     /* ...to hold public object returned by "om_get()"       */
```

The Main Program

This section describes the main program. Three calls usually precede any use of XDS.

First, **ds_initialize( )** is called to set up a *workspace*. A workspace is a memory area in which XDS can generate objects that will be used to pass information to the application. If the call is successful, it returns a handle that must be saved for the **ds_shutdown( )** call. If the call is unsuccessful, it returns NULL, but this example does not check for errors.

```
xdsWorkspace = ds_initialize();
```

If GDS is being used as the global directory service, the service packages are specified next. Packages consist of groups of objects, together with the associated supporting interface functionality, designed to be used for some specific end. This example uses the basic XDS service so DS_BASIC_DIR_CONTENTS_PKG is specified. The *featureList* parameter to **ds_version( )** is an array, not an object, since packages are not being handled yet:

```
dsStatus = ds_version(featureList, xdsWorkspace);
```

Note that, if you are *not* using GDS as your global directory service (in other words, if you are using XDS by itself), then do *not* call **ds_version( )**. If you use XDS to access the CDS namespace in any cell with an X.500 cell name, the **DS_BASIC_DIR_CONTENTS_PKG** is required.

From this point on, status is returned by XDS functions via a DS_status variable. DS_status is a handle to a private object, whose value is DS_SUCCESS (that is, NULL) if the call was successful. If something went wrong, the information in the (possibly complex) private error object has to be analyzed through calls to **om_get( )**, which is one of the general-purpose object management functions that belongs to XDS's companion interface XOM. Usage of **om_get( )** is demonstrated later on in this program, but return status is not checked in this example.

The third necessary call is to **ds_bind( )**. This call brings up the directory service, that binds to the CDS namespace. The DS_DEFAULT_SESSION parameter calls for a default session. The alternative is to build and fill out your own DS_C_SESSION object, specifying addresses, and pass that. The default is used in this example:

```
dsStatus = ds_bind(DS_DEFAULT_SESSION, xdsWorkspace,&session);
```

Reading a CDS Attribute

At this point, you can read a set of object attributes from the cell namespace entry. Call **ds_read( )** with the two objects that specify the entry to be read and the specific entry attribute you want:

```
dsStatus = ds_read(session,DS_DEFAULT_CONTEXT, Full_Entry_Name_Object,
          Entry_Info_Select_Object, &readResultObject,&dummy);
```

The DS_DEFAULT_CONTEXT parameter could be substituted with a DS_C_CONTEXT object, which would typically be reused during a series of related XDS calls. This object specifies and records how GDS should perform the operation, how much progress has been made in resolving a name, and so on.

If the call succeeds, the private object readResultObject contains a series of DS_C_ATTRIBUTE subobjects, one for each attribute read from the cell name entry. A complete recipe for the DS_C_READ_RESULT object can be found in "Chapter 10. XDS Class Definitions" on page 189, but the following is a skeletal outline of the object's structure:

```
DS_C_READ_RESULT
        DS_ENTRY: object(DS_C_ENTRY_INFO)
        DS_ALIAS_DEREFERENCED: OM_S_BOOLEAN
        DS_PERFORMER: object(DS_C_NAME)
    DS_C_ENTRY_INFO
```

```
            DS_FROM_ENTRY: OM_S_BOOLEAN
            DS_OBJECT_NAME: object(DS_C_NAME)
            DS_ATTRIBUTES: one or more object(DS_C_ATTRIBUTE)
        DS_C_NAME == DS_C_DS_DN
                DS_RDNS: object(DS_C_DS_RDN)

            DS_C_DS_RDN
                    DS_AVAS: object(DS_C_AVA)

                DS_C_AVA
                        DS_ATTRIBUTE_TYPE: OID string
                        DS_ATTRIBUTE_VALUES: anything

        DS_C_ATTRIBUTE —one for each attribute read
                DS_ATTRIBUTE_TYPE: OID string
                DS_ATTRIBUTE_VALUES: anything

        DS_C_ATTRIBUTE
                DS_ATTRIBUTE_TYPE: OID string
                DS_ATTRIBUTE_VALUES: anything
```

Figure 12 illustrates the general object structure of a `ADS_C_READ_RESULT`, showing only the object-valued attributes, and only one `DS_C_ATTRIBUTE` subobject.

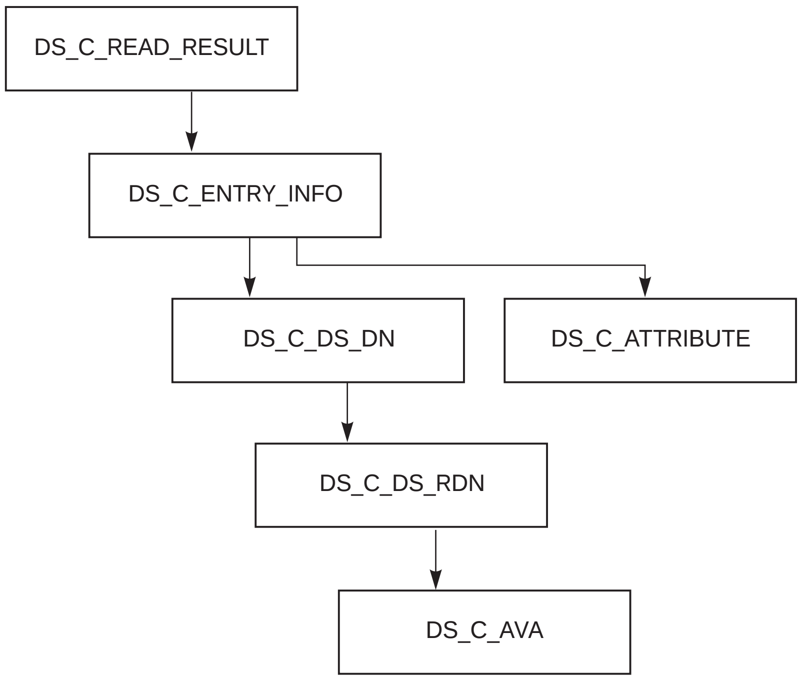


Figure 12. The DS_C_READ_RESULT Object Structure

Handling the Result Object

The next goal is to extract the instances of the `DS_C_ATTRIBUTE` subsubclass, one for each attribute read, from the returned object. The first step is to make a public copy of `readResultObject`, which is a *private* object, and therefore does not allow access to the object descriptors themselves. Using the XOM **om_get( )** function, you can make a public copy of `readResultObject`, and at the same time specify that only the relevant parts of it be preserved in the copy. Then, with a couple of calls to **om_get( )**, you can reduce the object to manageable size, leaving a superobject whose immediate subobjects are fairly easily accessed.

The **om_get( )** function takes as its third input parameter an OM_type_list, which is an array of OM_type. Possible parameters are DS_ENTRY, DS_ATTRIBUTES, DS_ATTRIBUTE_VALUES, and anything that can legitimately appear in an object descriptor's type field. The types specified in this parameter are interpreted according to the options specified in the preceding parameter. For example, the relevent attribute from the read result is DS_ENTRY. It contains the DS_C_ENTRY_INFO object, which in turn contains the DS_C_ATTRIBUTE objects. The DS_C_ATTRIBUTE objects contain the data read from the cell directory name entry. Therefore, you should specify the OM_EXCLUDE_ALL_BUT_THESE_TYPES option, which has the effect of excluding everything but the contents of the object's DS_ENTRY type attribute.

The OM_EXCLUDE_SUBOBJECTS option is also ORed into the parameter. Why would you not preserve the subobjects of DS_C_ENTRY_INFO? Because **om_get( )** works only on private, not on public, objects. If you were to use **om_get( )** on the entire object substructure, you would not be able to continue getting the subobjects, and instead you would have to follow the object pointers down to the DS_C_ATTRIBUTE. However, when **om_get( )** excludes subobjects from a copy, it does not really leave them out; it merely leaves the subobjects private, with a handle to the private objects where pointers would have been. This allows you to continue to call **om_get( )** as long as there are more subobjects.

The following is the first call:

```
/* The DS_C_READ_RESULT object that ds_read() returns has  */
/* one subobject, DS_C_ENTRY_INFO; it in turn has two sub- */
/* objects, that is a DS_C_NAME which holds the object's   */
/* distinguished name (which we don't care about here),    */
/* and a DS_C_ATTRIBUTE which contains the attribute info  */
/* we read; that one we want.  So we climb down to it ...  */
/* This om_get() will "return" the entry-info object ...   */

omStatus = om_get(readResultObject,
                  OM_EXCLUDE_ALL_BUT_THESE_TYPES +
                  OM_EXCLUDE_SUBOBJECTS,
                  I_want_entry_object,
                  OM_FALSE,
                  OM_ALL_VALUES,
                  OM_ALL_VALUES,
                  &entry,
                  &number_of_descriptors);
```

The number_of_descriptors parameter contains the number of attribute descriptors returned in the public copy, not in any excluded subobjects.

If an XOM function is successful, it returns an OM_SUCCESS code. Unsuccessful calls to XOM functions do not return error objects, but rather return simple error codes. The interface assumes that, if the XOM function does not accept your object, then you will not be able to get much information from any further objects. The return status is not checked in this example.

The return parameter entry should now contain a pointer to the DS_C_ENTRY_INFO object with the following immediate structure. (The number of instances of DS_ATTRIBUTES depends on the number of attributes read from the entry.)

```
DS_C_ENTRY_INFO
DS_FROM_ENTRY: OM_S_BOOLEAN
DS_OBJECT_NAME: object(DS_C_NAME)
```

```
DS_ATTRIBUTES: object(DS_C_ATTRIBUTE)
 DS_C_ATTRIBUTE
 DS_ATTRIBUTE_TYPE: OID string
 DS_ATTRIBUTE_VALUES: anything


DS_ATTRIBUTES: object(DS_C_ATTRIBUTE)
                      object(DS_C_ATTRIBUTE)
 DS_C_ATTRIBUTE
 DS_ATTRIBUTE_TYPE: OID string
 DS_ATTRIBUTE_VALUES: anything
```

The italics indicate private subobjects. Figure 13 shows the DS_C_ENTRY_INFO object. Only one instance of a DS_C_ATTRIBUTE subobject is shown in the figure; usually there are several such subobjects, all at the same level, each containing information about one of the attributes read from the entry. These subobjects are represented in DS_C_ENTRY_INFO as a series of descriptors of type DS_ATTRIBUTES, each of which has as its value a separate DS_C_ATTRIBUTE subobject.

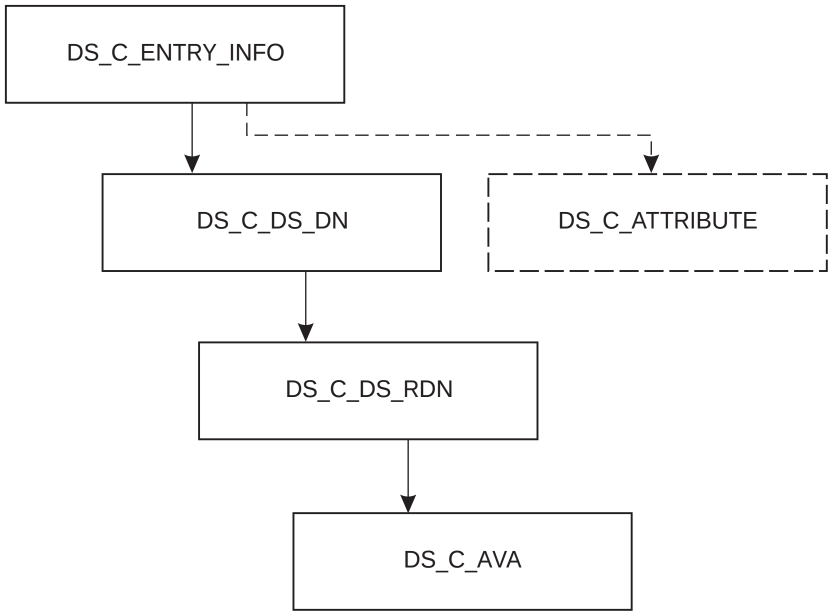


Figure 13. The DS_C_ENTRY_INFO Object Structure

Now extract the separate attribute values of the entry that was read. These were returned as separate object values of DS_ATTRIBUTES; each one has an object class of DS_C_ATTRIBUTE. To return any one of these subobjects, a second call to **om_get( )** is necessary, as follows:

```
/* The second om_get() returns one selected subobject     */
/* from the DS_C_ENTRY_INFO subobject we just got.  The    */
/* contents of "entry" as we enter this call is the       */
/* private subobject which is the value of DS_ATTRIBUTES.  */
/* If we were to make the following call with the         */
/* OM_EXCLUDE_SUBOBJECTS and without the                  */
/* OM_EXCLUDE_ALL_BUT_THESE_VALUES flags, we would get     */
/* back an object consisting of six private subobjects,    */
/* one for each of the attributes returned.  Note the      */
/* values for initial and limiting position: "2"          */
/* specifies that we want only the third DS_C_ATTRIBUTE    */
/* subobject to be gotten (the subobjects are numbered     */
/* from 0, not from 1), and the "3" specifies that we want */
/* no more than that--in other words, the limiting value   */
/* must always be one more than the initial value if the   */
/* latter is to have any effect.                           */
/* OM_EXCLUDE_ALL_BUT_THESE_VALUES is likewise required    */
/* for the initial and limiting values to have any         */
/* effect ...                                              */

omStatus = om_get(entry->value.object.object,
                  OM_EXCLUDE_ALL_BUT_THESE_TYPES
                  + OM_EXCLUDE_SUBOBJECTS
                  + OM_EXCLUDE_ALL_BUT_THESE_VALUES,
                  I_want_attribute_list,
                  OM_FALSE,
                  ((OM_value_position) 2),
                  ((OM_value_position) 3),
                  &entry,
                  &number_of_descriptors);
```

Note the value that is passed as the first parameter. Since **om_get( )** does not work on public objects, pass it the handle of the private subobject explicitly. To do this you have to know the arrangement of the descriptor's value union, which is defined in **xom.h**.

Representation of Object Values

The following is the layout of the `object` field in a descriptor's `value` union:

```
typedef struct {
OM_uint32       padding;
OM_object       object;
} OM_padded_object;
```

The following is the layout of the `value` union itself:

```
typedef union OM_value_union {
OM_string         string;
OM_boolean        boolean;
OM_enumeration    enumeration;
OM_integer        integer;
OM_padded_object  object;
} OM_value;
```

The following is the layout of the descriptor itself:

```
typedef struct OM_descriptor_struct {
OM_type                 type;
OM_syntax               syntax;
union OM_value_union    value;
} OM_descriptor;
```

Thus, if `entry` is a pointer to the `DS_C_ENTRY_INFO` object, then the private handle to the `DS_C_ATTRIBUTE` object you want next is the following:

`entry->value.object.object`

Extracting an Attribute Value

The last call yielded one separate `DS_C_ATTRIBUTE` subsubobject from the original returned result object:

```
DS_C_ATTRIBUTE
DS_ATTRIBUTE_TYPE: OID string
DS_ATTRIBUTE_VALUES: anything
```

Figure 14 illustrates what is left.

Figure 14. The DS_C_ATTRIBUTE Object Structure

A final call to **om_get( )** returns the single object descriptor that contains the actual value of the single attribute you selected from the returned object:

```
omStatus = om_get(entry->value.object.object,
                  OM_EXCLUDE_ALL_BUT_THESE_TYPES,
                  I_want_attribute_value,
                  OM_FALSE,
                  OM_ALL_VALUES,
                  OM_ALL_VALUES,
                  &entry,
                  &number_of_descriptors);
```

At this point, the value of `entry` is the base address of an object descriptor whose `entry->type` is `DS_ATTRIBUTE_VALUES`. Depending on the value found in `entry->syntax`, the value of the attribute can be read from `entry->value.string`, `entry->value.integer`, `entry->value.boolean`, or `entry->value.enumeration`.

For example, suppose the value of `entry->syntax` is `OM_S_OCTET_STRING`. The attribute value, represented as an octet string (*not* terminated by a NULL), is found in `entry->value.string.elements`; its length is found in `entry->value.string.length`.

You can check any attribute value against the value you get from the **cdscp** command by entering the following:

```
cdscp show object /.:/hosts/tamburlaine/self
```

For further information on **cdscp**, see the *IBM DCE Version 3.1 for AIX and Solaris: Administration Commands Reference*.

Note that you can always call **om_get( )** to get the *entire* returned object from an XDS call. This yields a full structure of object descriptors that you can manipulate like any other data structure. To do this with the **ds_read( )** return object would have required the following call:

```
/* make a public copy of ENTIRE object...   */

omStatus = om_get(readResultObject,
                  OM_NO_EXCLUSIONS,
                  ((OM_type_list) 0),
                  OM_FALSE,
                  ((OM_value_position) 0),
                  ((OM_value_position) 0),
                  &entry,
                  &number_of_descriptors);
```

At the end of every XDS session, you need to unbind from CDS and then deallocate the XDS and XOM structures and other storage. You must also explicitly deallocate any service-generated objects, whether public or private, with calls to **om_delete( )**, as follows:

```
/* delete service-generated public or private objects... */

omStatus = om_delete(readResultObject);
omStatus = om_delete(entry);

/* unbind from the CDS...  */
dsStatus = ds_unbind(session);

/* close down the workspace... */
dsStatus = ds_shutdown(xdsWorkspace);

exit();
```

Creating New CDS Entry Attributes

The following subsections provide the procedure and some code examples for creating new CDS entry attributes.

Procedure for Creating New Attributes

To create new attributes of your own on cell namespace entries, you must do the following:

1. Allocate a new ISO OID for the new attribute. For information on how to do this, see "Chapter 2. Programming in the CDS Namespace" on page 17 of this guide and the *IBM DCE Version 3.1 for AIX and Solaris: Administration Guide*.
2. Enter the new attribute's name and OID in the file **/.:/opt/dcelocal/etc/cds_attributes**. This text file contains OID-to-readable string mappings that are used, for example, by **cdscp** when it displays CDS entry attributes. Each entry also gives a syntax for reading the information in the entry itself. This should be congruent with the format of the data you intend to write in the attribute. For more information about the `cds_attributes` file, see the *IBM DCE Version 3.1 for AIX and Solaris: Administration Guide*.
3. In the **xdscds.h** header file, define an appropriate OID string constant to represent the new attribute.

   For example, the following shows the **xdscds.h** definition for the CDS `CDS_Class` attribute:

   ```
   #define OMP_O_DSX_A_CDS_Class     "\x2B\x16\x01\x03\x0F"
   ```

   Note the XDS internal form of the name. This is what `DSX_A_CDS_Class` looks like when it has been exported using `OM_EXPORT` in an application, as all OIDs must be. Thus, if you wanted to create a CDS attribute called `CDS_Brave_New_Attrib`, you would obtain an OID from your administrator and add the following line to **xdscds.h**:

   ```
   #define OMP_O_DSX_A_CDS_Brave_New_Attrib "your_OID"
   ```
4. In an application, call the XDS **ds_modify_entry( )** routine to add the attribute to the cell namespace entry of your choice.

Rules for Transforming an OID into XDS Internal Form

In Item 3 in the previous procedure, the **CDS_Class** attribute is shown in the XDS internal form as `\x2B\x16\x01\x03\x0F`. In the cds_attributes file, the OID is defined as `1.3.22.1.3.15`. The following provides the rules for transforming an OID into the XDS internal form. They must be applied in the sequence listed below.

1. The first two OID values are collapsed into a single byte using the following formula:

   OID Value 1 x 40 + OID Value 2

   For example, OID 1.3 transforms to `\x2B (1 x 40 + 3 = 43 or \x2B`.
2. If an OID value is 127 or less, it is represented by its hex value. For example, the OID value 22 above is represented by `\x16`.
3. If an OID value is larger than 127, the value is represented by multiple bytes in the XDS internal form. In the XDS internal form, the high order bit (`\x80`) means that the following byte is part of the same OID value. The low order 7 bits are concatenated with the lower 7 bits of the next byte. This concatenation continues until a byte is reached that is `\x7F` or less. Once this transformation is complete, the values of the bits concatenated together are the value of the OID.

   For example, the OID for the Service Package is `1.3.12.2.1011.28.0` and has an XDS internal representation of `\x2B\x0C\x02\x87\x73\x1C\00`. The following illustrates the mapping used:

   ```
       1.3 -> \x2B       (Rule 1)
        12 -> \x0C       (Rule 2)
         2 -> \x02       (Rule 2)
      1011 -> \87\x73    (Rule 3 -- see below)
        28 -> \x1C       (Rule 2)
         0 -> \x00       (rule 2)
   ```

   If you concatenate the last 7 bits of `\x87` and `\x73`, the value `\x03\xF3` is obtained (which is `1011` in decimal).

Coding Examples

In the following code fragments, a set of declarations similar to those in the previous examples is assumed.

The **ds_modify_entry( )** function, which is called to add new attributes to an entry or to write new values into existing attributes, requires a `DS_C_ENTRY_MOD_LIST` input object whose contents specify the attributes and values to be written to the entry. The name, as always, is specified in a `DS_C_DS_DN` object. The following is a static declaration of such a list, which consists of two attributes:

```
static OM_descriptor    Entry_Modification_Object_1[] = {
  OM_OID_DESC(OM_CLASS, DS_C_ENTRY_MOD),
  OM_OID_DESC(DS_ATTRIBUTE_TYPE, DSX_A_CDS_Brave_New_Attrib),
  {DS_ATTRIBUTE_VALUES, OM_S_PRINTABLE_STRING,
OM_STRING("O brave new attribute")},
  {DS_MOD_TYPE, OM_S_ENUMERATION, DS_ADD_ATTRIBUTE},
  OM_NULL_DESCRIPTOR
};

static OM_descriptor    Entry_Modification_Object_2[] = {
  OM_OID_DESC(OM_CLASS, DS_C_ENTRY_MOD),
  OM_OID_DESC(DS_ATTRIBUTE_TYPE, DSX_A_CDS_Class),
  {DS_ATTRIBUTE_VALUES, OM_S_PRINTABLE_STRING, \
      OM_STRING("Miscellaneous")},
  {DS_MOD_TYPE, OM_S_ENUMERATION, DS_ADD_ATTRIBUTE},
```

```
  OM_NULL_DESCRIPTOR
};

static OM_descriptor    Entry_Modification_List_Object[] = {
  OM_OID_DESC(OM_CLASS, DS_C_ENTRY_MOD_LIST),
  {DS_CHANGES, OM_S_OBJECT, {0, Entry_Modification_Object_1}},
  {DS_CHANGES, OM_S_OBJECT, {0, Entry_Modification_Object_2}},
  OM_NULL_DESCRIPTOR
};
```

A full description of this object can be found in “XDS/CDS Object Recipes” on page 63. There could be any number of additional attribute changes in the list; this would mean additional DS_C_ENTRY_MOD objects declared, and an additional DS_CHANGES descriptor declared and initialized in the DS_C_ENTRY_MOD_LIST object.

With the DS_C_ENTRY_MOD_LIST class object having been declared as shown previously, the following code fragment illustrates how to call XDS to write a new attribute value (actually two new values since two attributes are contained in the list object). Note that any of the attributes may be new, although the entry itself must already exist.

```
dsStatus = ds_modify_entry(session,    /* Directory session */
                                     /* from "ds_bind()"       */
         DS_DEFAULT_CONTEXT, /* Usual directory context        */
         Full_Entry_Name_Object, /* Entry name object          */
         Entry_Modification_List_Object, /* Entry Modification */
                                         /* object             */
                 &dummy);        /* Unsupported argument
*/
```

If the entire entry is new, you must call **ds_add_entry( )**. This function requires an input object of class DS_C_ATTRIBUTE_LIST, whose contents specify the attributes (and values) to be attached to the new entry. Following is the static declaration for an attribute list that contains three attributes:

```
static OM_descriptor
Class_Attribute_Object[] = {
     OM_OID_DESC(OM_CLASS, DS_C_ATTRIBUTE),
     OM_OID_DESC(DS_ATTRIBUTE_TYPE, DSX_A_CDS_Class),
     {DS_ATTRIBUTE_VALUES, OM_S_PRINTABLE_STRING, OM_STRING("Printer")},
     OM_NULL_DESCRIPTOR
};

static OM_descriptor    ClassVersion_Attribute_Object[] = {
     OM_OID_DESC(OM_CLASS, DS_C_ATTRIBUTE),
     OM_OID_DESC(DS_ATTRIBUTE_TYPE, DSX_A_CDS_ClassVersion),
     {DS_ATTRIBUTE_VALUES, OM_S_PRINTABLE_STRING, OM_STRING("1.0")},
     OM_NULL_DESCRIPTOR
};
static OM_descriptor    My_Own_Attribute_Object[] = {
     OM_OID_DESC(OM_CLASS, DS_C_ATTRIBUTE),
     OM_OID_DESC(DS_ATTRIBUTE_TYPE, DSX_A_CDS_My_OwnAttribute),
     {DS_ATTRIBUTE_VALUES, OM_S_PRINTABLE_STRING, OM_STRING("zorro")},
     OM_NULL_DESCRIPTOR
};

static OM_descriptor    Attribute_List_Object[] = {
     OM_OID_DESC(OM_CLASS, DS_C_ATTRIBUTE_LIST),
     {DS_ATTRIBUTES, OM_S_OBJECT, {0, Class_Attribute_Object}},
     {DS_ATTRIBUTES, OM_S_OBJECT, {0, ClassVersion_Attribute_Object}},
     {DS_ATTRIBUTES, OM_S_OBJECT, {0, My_Own_Attribute_Object}},
     OM_NULL_DESCRIPTOR
};
```

The **ds_add_entry( )** function also requires a DS_C_DS_DN class object containing the new entry's full name, for example:

```
/.../osf.org.dce/subsys/doc/my_book
```

where every member of the name exists except for the last one, my_book. Assuming that Full_Entry_Name_Object is a DS_C_DS_DN object, the following code shows what the call would look like:

```
dsStatus = ds_add_entry(session,    /* Directory session   */
                                      /* from "ds_bind()"      */
           DS_DEFAULT_CONTEXT,  /* Usual directory context     */
           Full_Entry_Name_Object,    /* Name of new entry     */
           Attribute_List_Object,     /* Attributes to be      */
                     /* attached to new entry, with values     */
           &dummy);               /* Unsupported argument
*/
```

Object-Handling Techniques

The following subsections describe the use of XOM and discuss dynamic object creation.

Using XOM to Access CDS

The following code fragments demonstrate an alternative way to set up the entry modification object for a **ds_modify_entry( )** call, mainly for the sake of showing how the **om_put( )** and **om_write( )** functions are used.

The following technique is used to initialize the modification object:

1. The **om_create( )** function is called to generate a private object of a specified class.
2. The **om_put( )** function is called to copy statically declared attributes into a declared private object.
3. The **om_write( )** function is called to write the value string, which is to be assigned to the attribute, into the private object.
4. The **om_get( )** function is called to make the private object public.
5. The object is now public, and its address is inserted into the DS_C_ENTRY_MOD_LIST object's DS_CHANGES attribute.

The following new declarations are necessary:

```
OM_private_object newAttributeMod_priv;
     /* ...handle to a private object to "om_put()" to              */

OM_public_object newAttributeMod_pub;
     /* ...to hold public object from "om_get()"                    */

OM_type types_to_include[] = {DS_ATTRIBUTE_TYPE, DS_ATTRIBUTE_VALUES,
                              DS_MOD_TYPE, OM_NO_MORE_TYPES};
     /* ...that is, all attribute values of the Entry Modification    */
     /*  object.  For "om_put()" and "om_get()"                       */

char *my_string = "O brave new attribute";
     /* ...value I want to write into attribute                       */
```

```
OM_value_position number_of_descriptors;
     /* ...to hold value returned by "om_get()"                     */
```

First, use XOM to generate a private object of the desired class:

```
omStatus = om_create(DS_C_ENTRY_MOD,  /*Class of object              */
                 OM_TRUE,   /* Initialize attributes per defaults     */
                 xdsWorkspace,  /* Our workspace handle               */
                 &newAttributeMod_priv);  /* Created object handle  */
```

Next, copy the public object's attributes into the private object:

```
omStatus = om_put(newAttributeMod_priv,/* Private object to copy     */
                                        /*  attributes into           */
                 OM_REPLACE_ALL, /* Which attributes to replace in    */
                                  /*  destination object              */
                 Entry_Modification_Object, /* Source object to copy  */
                                             /*  attributes from      */
                 types_to_include, /* List of attribute types we      */
                                    /* want copied                    */
                 0, 0); /* Start-stop index for multivalued           */
                         /*  attributes; ignored with OM_REPLACE_ALL */
```

Since **om_put( )** ignores the class of the source object (the object from which attributes are being copied), it is not necessary to declare class descriptors for the source objects. In other words, the static declarations could have omitted the `OM_CLASS` initializations if this technique were being used, for example:

```
static OM_descriptor    Entry_Modification_Object_2[]= {

/*     OM_OID_DESC(OM_CLASS, DS_C_ENTRY_MOD),              */
/*     Not needed for "om_put()" ...                       */

     OM_OID_DESC(DS_ATTRIBUTE_TYPE, DSX_A_CDS_Class),
     {DS_ATTRIBUTE_VALUES, OM_S_PRINTABLE_STRING, \
       OM_STRING("Miscellaneous")},
     {DS_MOD_TYPE, OM_S_ENUMERATION, DS_ADD_ATTRIBUTE},
     OM_NULL_DESCRIPTOR
};
```

The `OM_CLASS` was already properly initialized by **om_create( )**.

Next, write the attribute value string into the private object:

```
omStatus =om_write(newAttributeMod_priv,/* Private object to write to */
                    DS_ATTRIBUTE_VALUES, /* Attribute type whose value*/
                                         /*  we're writing            */
                    0, /* Descriptor index if attribute is multivalued*/
                    OM_S_PRINTABLE_STRING, /* Syntax of value         */
                    0, /* Offset in source string to write from       */
                    my_string); /* Source string to write from        */
```

Now make the whole thing public again:

```
omStatus = om_get(newAttributeMod_priv,/* Private object to get       */
                  0,                    /* Get everything             */
                  types_to_include,     /* All attribute types        */
                  0,                    /* Unsupported argument       */
                  0, 0, /* Start-stop descriptor index for multival-  */
                        /*  ued attributes; ignored in this case      */
                  &newAttributeMod_pub, /* Pointer to returned copy */
                  &number_of_descriptors); /* Number of attribute   */
                                           /*  descriptors returned   */
```

Finally, insert the address of the subobject into its superobject:

```
Entry_Modification_List_Object[1].value.object.object = \
  newAttributeMod_pub;
```

Dynamic Creation of Objects

Objects can be completely dynamically allocated and initialized; however, you have to implement the routines to do this yourself. The examples in this section are code fragments. For complete examples, see the samples located in the **/opt/declocal/examples/xdsxom** directory.

Initialization of object structures can be automated by declaring macros or functions to do this. For example, the following macro initializes one object descriptor with a full set of appropriate values:

```
/* Put a C-style (NULL-terminated) string into an object and */
/* set all the other descriptor fields to requested values   */
#define FILL_OMD_STRING( desc, index, typ, syntx, val ) \
        desc[index].type = typ; \
        desc[index].syntax = syntx; \
        desc[index].value.string.length = \
          (OM_element_position)strlen(val); \
        desc[index].value.string.elements = val;
```

When generating objects, use **malloc( )** to allocate space for the number of objects desired, and then use macros (or functions) such as the preceding one to initialize the descriptors. The following code fragment shows how this can be done for the top-level object of a DS_C_DS_DN object, such as the one described near the beginning of this chapter. Recall that DS_C_DS_DN has a separate DS_RDNS descriptor for each name piece in the full name.

```
/* Calculate number of "DS_RDNS"attributes there should be ... */
numberOfPieces =  number_of_name_pieces;

/* Allocate space for that many descriptors, plus one for the   */
/* object class at the front, and a NULL descriptor at the back */

Name_Object = (OM_object)malloc((numberOfPieces + 2) \
 * sizeof(OM_descriptor));
if(Name_Object == NULL)                    /* "malloc()" failed */
return OM_MEMORY_INSUFFICIENT;

/* Initialize it as a DS_C_DS_DN object by placing that class   */
/*  identifier in the first position...                         */

FILL_OMD_XOM_STRING(Name_Object, 0, OM_CLASS,
    OM_S_OBJECT_IDENTIFIER_STRING,
DS_C_DS_DN)
```

Note that all these steps would have to be repeated for each of the DS_C_DS_RDN objects required as attribute values of the DS_C_DS_DN. Then a tier of DS_C_AVA objects would have to be created in the same way, since each of the DS_C_DS_RDNs requires one of them as *its* attribute value.

You could now use **om_create( )** and **om_put( )** to generate a private copy of this object, if so desired.

The application is responsible for managing the memory it allocates for such dynamic object creation.

XDS/CDS Object Recipes

The following subsections contain shorthand for object classes. For example, if you look at the reference pages for the `ds_ *( )` functions, you will see that an object of class `DS_C_NAME` is required to hold entry names you want to pass to the call, *not* `DS_C_DS_DN` as is stated in this chapter. However, `DS_C_NAME` is in fact an abstract class with only one subclass `DS_C_DS_DN` so, in this chapter, `DS_C_DS_DN` is used.

Input XDS/CDS Object Recipes

In general, the objects you work with in an XDS/CDS application fall into two categories:

- Objects you have to supply as *input parameters* to XDS functions
- Objects returned to you as *output* by XDS functions

This section describes only the first category, since you have to construct these input objects yourself.

Table 4 shows XDS functions and the objects given to them as input parameters.

Only items significant to CDS are listed in the table. `DS_C_SESSION` and `DS_C_CONTEXT` are ignored. `DS_C_SESSION` is returned by **ds_bind( )**, which usually receives the `DS_DEFAULT_SESSION` constant as input. `DS_C_CONTEXT` is usually substituted by the `DS_DEFAULT_CONTEXT` constant.

Note: `DS_C_NAME` is an abstract class that has the single subclass `DS_C_DS_DN`. Therefore, `DS_C_NAME` is practically the same thing as `DS_C_DS_DN`.

Table 4. Directory Service Functions With Their Required Input Objects

Function	Input Object
ds_add_entry( )	DS_C_NAME
	DS_C_ATTRIBUTE_LIST
ds_bind( )	None
ds_compare( )	DS_C_NAME
	DS_C_AVA
ds_initialize( )	None
ds_list( )	DS_C_NAME
ds_modify_entry( )	DS_C_NAME
	DS_C_ENTRY_MOD_LIST
ds_read( )	DS_C_NAME
	DS_C_ENTRY_INFO_SELECTION
ds_remove_entry( )	DS_C_NAME
ds_shutdown( )	None

Table 4. Directory Service Functions With Their Required Input Objects (continued)

Function	Input Object
ds_unbind( )	None
ds_version( )	None

Input Object Classes for XDS/CDS Operations

The following subsections contain information about all the object types required as input to any of the XDS functions that can be used to access CDS. In order to use these functions successfully, you must be able to construct and modify the objects that the functions expect as their input parameters. XDS functions require most of their input parameters to be wrapped in a nested series of data structures that represent objects, and these functions deliver their output returns to callers in the same object form.

Objects that are returned to you by the interface are not difficult to manipulate because the **om_get( )** function allows you to go through them and retrieve only the value parts you are interested in, and discard the parts of data structures you are not interested in. However, any objects you are required to supply as *input* to an XDS or XOM function are another matter: you must build and initialize these object structures yourself.

The basics of object building have already been explained earlier in this chapter. Each object described in the following subsections is accompanied by a static declaration in C of a very simple instance of that object class. The objects in an application are usually built dynamically (this technique was demonstrated earlier in this chapter). The static declarations that follow give a simple example of what the objects look like.

An object's properties, such as what sort of values it can hold, how many of them it can hold, and so on, are determined by the *class* the object belongs to. Each class consists of one or more *attributes* that an object can have. The attributes hold whatever values the object contains. Thus, the objects are data structures that all look the same (and can be handled in the

same way) from the outside, but whose specific data fields are determined by the class each object belongs to. At the abstract level, objects consist of attributes, just as structures consist of fields.

XDS/CDS Object Types

Following is a list of all the object types that are described in the following subsections. Most of these objects are object structures; that is, compounds consisting of superobjects that contain subobjects as some of their values. These subobjects may in turn contain other objects, and so on. Subobjects are indicated by indentation. A `DS_C_DS_DN` object contains at least one `DS_C_DS_RDN` object, and each `DS_C_DS_RDN` contains one `DS_C_AVA` object. Note that subobjects can, and often do, exist by themselves, depending on what object class is called for by a given function. This list contains all the possible kinds of objects that can be required as input for any XDS/CDS operation.

- `DS_C_ATTRIBUTE_LIST`
  - `DS_C_ATTRIBUTE`
- `DS_C_DS_DN`
  - `DS_C_DS_RDN`

  - DS_C_AVA
- DS_C_ENTRY_MOD_LIST
  - DS_C_ENTRY_MOD
- DS_C_ENTRY_INFO_SELECTION

In each section, information is provided for the described object's attributes. All its attributes are listed.

The illustrations in the following sections can be compared to the same object classes' tabular definitions later in this guide.

The DS_C_ATTRIBUTE_LIST Object

A DS_C_ATTRIBUTE_LIST class object is required as input to **ds_add_entry( )**. The object contains a list of the directory attributes you want associated with the entry that is to be added.

Its general structure is as follows:

- Attribute List class type attribute
- Zero or more Attribute objects:
  - Attribute class type attribute
  - Attribute Type attribute
  - Zero or more Attribute Value(s)

Thus, a DS_C_ATTRIBUTE_LIST object containing one attribute consists of two object descriptor arrays because each additional attribute in the list requires an additional descriptor array to represent it. The subobject arrays' names (that is, addresses) are the contents of the value fields in the DS_ATTRIBUTES object descriptors.

Figure 15 on page 66 shows the attributes of the DS_C_ATTRIBUTE_LIST object.

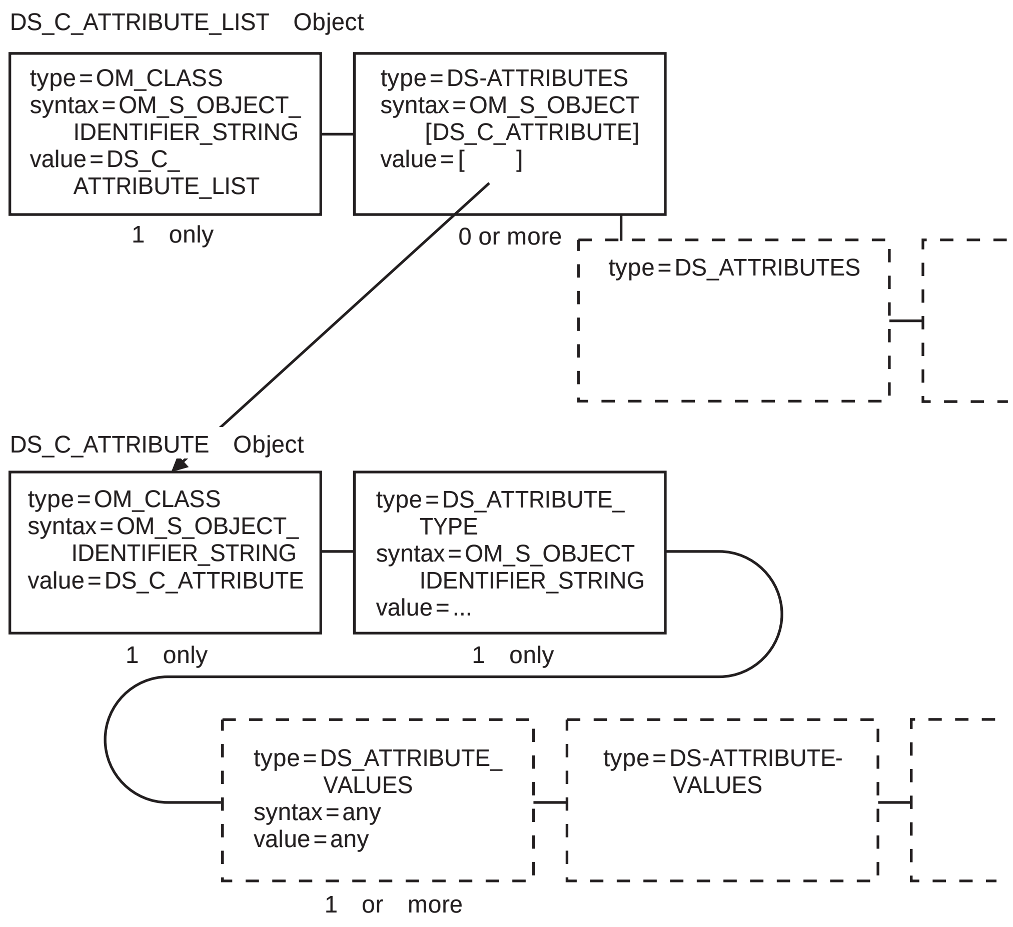


Figure 15. The DS_C_ATTRIBUTE_LIST Object

- `OM_CLASS`

  The value of this attribute is an OID string that identifies the object's class; its value is always `DS_C_ATTRIBUTE_LIST`.
- `DS_ATTRIBUTES`

  This is an attribute whose value is another object of class `DS_C_ATTRIBUTE` (see "The DS_C_ATTRIBUTE Object"). The attribute is defined by a separate array of object descriptors whose base address is the value of the `DS_ATTRIBUTES` attribute. Note that there can be any number of instances of this attribute and, therefore, any number of subobjects.

The DS_C_ATTRIBUTE Object

An object of this class can be an attribute of a `DS_C_ATTRIBUTE_LIST` object (see "The DS_C_ATTRIBUTE_LIST Object" on page 65).

- `OM_CLASS`

  The value of this attribute is an OID string that identifies the object's class; its value is always `DS_C_ATTRIBUTE`.
- `DS_ATTRIBUTE_TYPE`

  The value of this attribute, which is an OID string, identifies the directory attribute whose value is contained in this object.
- `DS_ATTRIBUTE_VALUES`

These are the actual values for the directory attribute represented by this DS_C_ATTRIBUTE object. Both the value syntax and the number of values depend on what directory attribute this is; that is, they depend on the value of DS_ATTRIBUTE_VALUE.

Example Definition of a DS_C_ATTRIBUTE_LIST Object

The following code fragment is a definition of a DS_C_ATTRIBUTE_LIST object.

```
static OM_descriptor    Single_Attribute_Object[] = {
 OM_OID_DESC(OM_CLASS, DS_C_ATTRIBUTE),
 OM_OID_DESC(DS_ATTRIBUTE_TYPE, DSX_A_CDS_Class),
 {DS_ATTRIBUTE_VALUES, OM_S_PRINTABLE_STRING, \
   OM_STRING("Printer")},
 OM_NULL_DESCRIPTOR
};
static OM_descriptor    Attribute_List_Object[] = {
 OM_OID_DESC(OM_CLASS, DS_C_ATTRIBUTE_LIST),
 {DS_ATTRIBUTES, OM_S_OBJECT, {0, Single_Attribute_Object}},
 OM_NULL_DESCRIPTOR
};
```

The DS_C_DS_DN Object

DS_C_DS_DN class objects are used to hold the full names of directory entries (distinguished names). You need an object of this class to pass directory entry names to the following XDS functions:

- **ds_add_entry( )**
- **ds_compare( )**
- **ds_list( )**
- **ds_modify_entry( )**
- **ds_read( )**
- **ds_remove_entry( )**

Figure 16 on page 68 shows the attributes of a DS_C_DS_DN object.

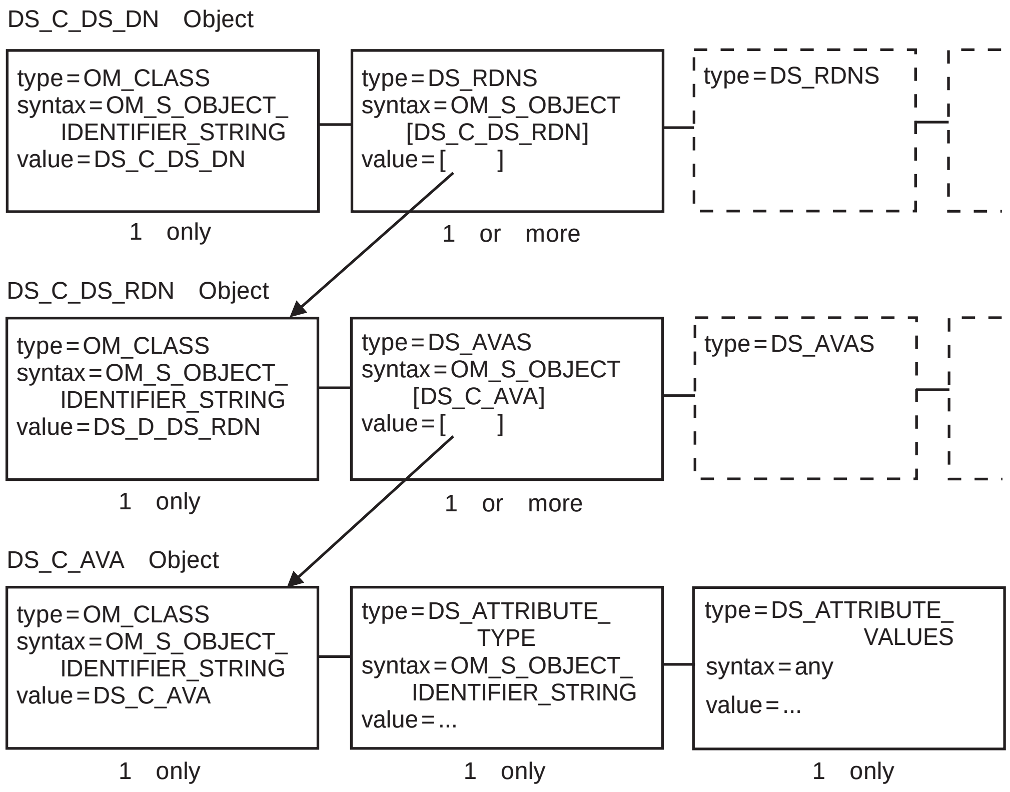


Figure 16. DS_C_DS_DN Object Attributes

- `OM_CLASS`

  The value of this attribute is an OID string that identifies the object's class; its value is `DS_C_DS_DN`.
- `DS_RDNS`

  This is an attribute whose value is another object of class `DS_C_DS_RDN` (see "The DS_C_DS_RDN Object" on page 69). The `DS_C_DS_RDN` object is defined by a separate array of object descriptors whose base address is the value of the `DS_RDNS` attribute.

  There are as many `DS_RDNS` attributes in a `DS_C_DS_DN` object as there are separate name components in the full directory entry name. For example, suppose you wanted to represent the following CDS entry name:

  ```
  /.../C=US/O=OSF/OU=DCE/hosts/brazil/self
  ```

  This would require a total of six instances of the `DS_RDNS` attribute in the `DS_C_DS_DN` object. The ***/.../*** (global root prefix) is not represented. This means that another six object descriptor arrays are required to hold the RDN objects, as well as six object descriptors in the present object, one to hold (as the value of a `DS_RDNS` attribute) a pointer to each array.

  Note that the order of these `DS_RDNS` attributes is significant; that is, the first `DS_RDNS` should contain as its value a pointer to the array representing the `C=US` part of the name; the next `DS_RDNS` should contain as its value a pointer to the array representing the `O=OSF` part, and so on. The root part of the name is not represented at all.

The DS_C_DS_RDN Object

DS_C_DS_RDN class objects are required as values for the DS_RDNS attributes of DS_C_DS_DN objects. (For an illustration of its structure, see Figure 16 on page 68.) RDN refers to the X.500 term RDN that is used to signify a part of a full entry name. Separate objects of this class are not usually required as input to XDS functions.

The standard permits multiple AVAs in an RDN, but the DCE Directory and XDS API restrict an RDN to one AVA.

- OM_CLASS

  The value of this attribute is an OID string that identifies the object's class; its value is always DS_C_DS_RDN.
- DS_AVAS

  This is an attribute whose value is yet another object of class DS_C_AVA (see "The DS_C_AVA Object"). The DS_C_AVA object is defined by a separate array of object descriptors whose base address is the value of the DS_AVAS attribute.

  Note that there can only be one instance of this attribute in the DS_C_RDN object. The object descriptor array describing this object always consists of three object descriptor structures: the first describes the object's class, the second describes the DS_AVAS attribute, and the third descriptor is the terminating NULL.

The DS_C_AVA Object

The DS_C_AVA class object is used to hold an actual value. The value is usually in the form of one of the many different XOM string types. (For an illustration of its structure, see Figure 16 on page 68.)

In calls to **ds_compare( )**, an object of this type is required to hold the type and value of the attribute that you want compared with those in the entry you specify. It holds the type and value in a separate DS_C_DS_DN object.

DS_C_AVA is also included here because it is a required subsubobject of DS_C_DS_DN itself. DS_C_AVA is the subobject in which the name part's actual literal value is held.

- OM_CLASS

  The value of this attribute is an OID string that identifies the object's class; its value is always DS_C_AVA.
- DS_ATTRIBUTE_TYPE

  The value of this attribute, which is an OID string, identifies the directory attribute whose value is contained in this object.
- DS_ATTRIBUTE_VALUES

  This is the literal value of what is represented by this DS_C_AVA object.

  If the DS_C_AVA object is a subobject of DS_C_DS_RDN (and therefore also of DS_C_DS_DN), then the value is a string representing the part of the directory entry name represented by this object. For example, if the DS_C_DS_RDN object contains the O=OSF part of an entry name, then the string OSF is the value of the DS_ATTRIBUTE_VALUES attribute, and DS_A_COUNTRY_NAME is the value of the DS_ATTRIBUTE_TYPE attribute.

  On the other hand, if DS_C_AVA contains an entry attribute type and value to be passed to **ds_compare( )**, then DS_ATTRIBUTE_TYPE identifies the type of the attribute, and DS_ATTRIBUTE_VALUES contains a value, which is appropriate for the attribute type, to be compared with the entry value.

For example, suppose you wanted to compare a certain value with a CDS entry's CDS_Class attribute's value. The identifiers for all the valid CDS entry attributes are located in the file **/.:/opt/dcelocal/etc/cds_attributes**. The value of DS_ATTRIBUTE_TYPE would be CDS_Class, which is the label of an object identifier string, DS_ATTRIBUTE_VALUES would contain some desired value in the correct syntax for ACDS_Class. The syntax is also found in the cds_attributes file; for CDS_Class it is byte; that is, a character string. The user could code **OM_S_OCTET_STRING** for **DS_ATTRIBUTE_VALUES**. See "Attribute and Data Type Translation" on page 74.

Example Definition of a DS_C_DS_DN Object

The following code fragment shows an example definition for a DS_C_DS_DN object.

```
static OM_descriptor    Entry_String_Object[] = {
 OM_OID_DESC(OM_CLASS, DS_C_AVA),
 OM_OID_DESC(DS_ATTRIBUTE_TYPE, DSX_TYPELESS_RDN),
 {DS_ATTRIBUTE_VALUES, OM_S_PRINTABLE_STRING, \
   OM_STRING("brazil")},
 OM_NULL_DESCRIPTOR
};

static OM_descriptor    Entry_Part_Object[] = {
 OM_OID_DESC(OM_CLASS, DS_C_DS_RDN),
 {DS_AVAS, OM_S_OBJECT, {0, Entry_String_Object}},
 OM_NULL_DESCRIPTOR
};

static OM_descriptor    Entry_Name_Object[] = {
 OM_OID_DESC(OM_CLASS, DS_C_DS_DN),
 {DS_RDNS, OM_S_OBJECT, {0, Entry_Part_Object}},
 OM_NULL_DESCRIPTOR
};
```

The DS_C_ENTRY_MOD_LIST Object

DS_C_ENTRY_MOD_LIST class objects, which contain a list of changes to be made to some directory entry, must be passed to **ds_modify_entry( )**. DS_C_ENTRY_MOD_LIST objects have the attributes shown in Figure 17 on page 71.

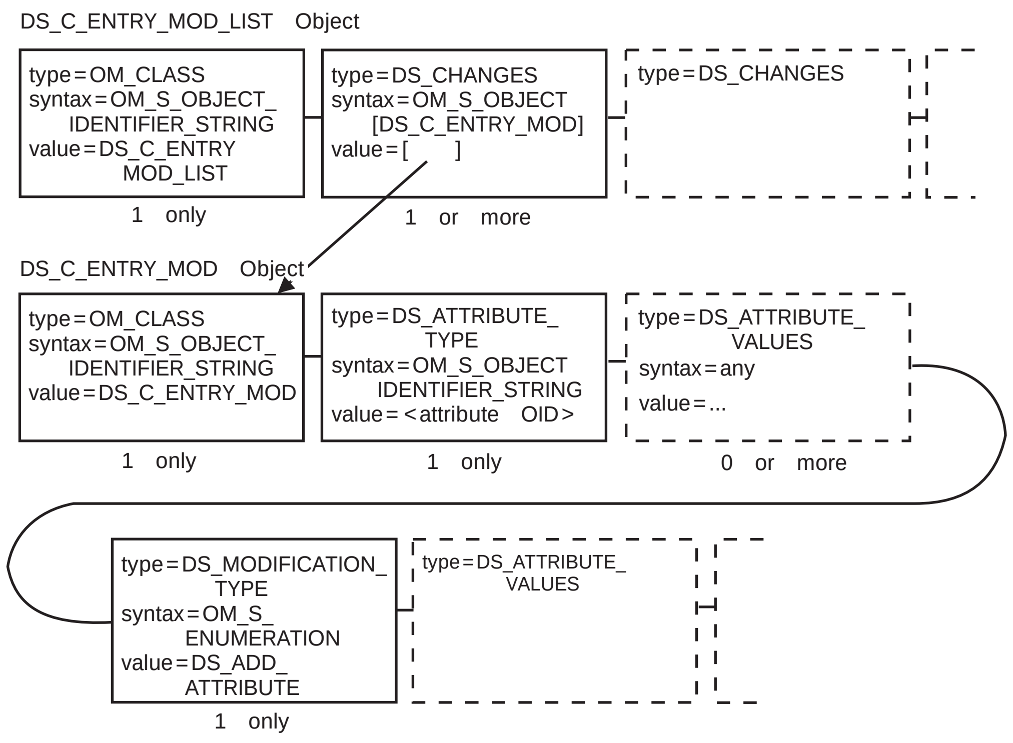

DS_C_ENTRY_MOD_LIST Object
type=OM_CLASS
syntax=OM_S_OBJECT_ IDENTIFIER_STRING
value=DS_C_ENTRY MOD_LIST
1 only
type=DS_CHANGES
syntax=OM_S_OBJECT [DS_C_ENTRY_MOD]
value=[ ]
1 or more
type=DS_CHANGES
DS_C_ENTRY_MOD Object
type=OM_CLASS
syntax=OM_S_OBJECT_ IDENTIFIER_STRING
value=DS_C_ENTRY_MOD
1 only
type=DS_ATTRIBUTE_ TYPE
syntax=OM_S_OBJECT IDENTIFIER_STRING
value=<attribute OID>
1 only
type=DS_ATTRIBUTE_ VALUES
syntax=any
value=...
0 or more
type=DS_MODIFICATION_ TYPE
syntax=OM_S_ ENUMERATION
value=DS_ADD_ ATTRIBUTE
1 only
type=DS_ATTRIBUTE_ VALUES

Figure 17. The DS_C_ENTRY_MOD_LIST Object

- OM_CLASS

  The value of this attribute is an OID string that identifies the object's class; its value is always DS_C_ENTRY_MOD_LIST.

- DS_CHANGES

  This is an attribute whose value is another object of class DS_C_ENTRY_MOD (see Section 3.5.2.10). The DS_C_ENTRY_MOD object is defined by a separate array of object descriptors whose base address is the value of the DS_CHANGES attribute.

  Note that there can be one or more instances of this attribute in the object, which is why it is called _LIST. Each attribute contains one separate entry modification. To learn how the modification itself is specified, see "The DS_C_ENTRY_MOD Object". The order of multiple instances of this attribute is significant because, if more than one modification is specified, the modifications are performed by **ds_modify_entry( )** in the order in which the DS_CHANGES attributes appear in the DS_C_ENTRY_MOD_LIST object.

The DS_C_ENTRY_MOD Object

The DS_C_ENTRY_MOD class object holds the information associated with a directory entry modification. (For an illustration of its structure, see Figure 17.) Each DS_C_ENTRY_MOD object describes one modification. To create a list of modifications suitable to be passed to a **ds_modify_entry( )** call, describe each modification in a separate DS_C_ENTRY_MOD object, and then insert these objects as multiple instances of the DS_CHANGES attribute in a DS_C_ENTRY_MOD_LIST object (see "The DS_C_ENTRY_MOD_LIST Object" on page 70).

- OM_CLASS

  The value of this attribute is an OID string that identifies the object's class; its value is always DS_C_ENTRY_MOD.

- DS_ATTRIBUTE_TYPE

The value of this attribute, which is an OID string, identifies the directory attribute whose modification is described in this object.

- DS_ATTRIBUTE_VALUES

  These are the values required for the entry modification; their type and number depend on both the entry type and the modification requested.
- DS_MOD_TYPE

  The value of this attribute identifies the kind of modification requested. It can be one of the following:
  - DSA_ADD_ATTRIBUTE

    The attribute specified by DS_ATTRIBUTE_TYPE is not currently in the entry. It should be added, along with the value(s) specified by DS_ATTRIBUTE_VALUES, to the entry. The entry itself is specified in a separate DS_C_DS_DN object, which is also passed to **ds_modify_entry( )**.
  - DS_ADD_VALUES

    The specified attribute is currently in the entry. The value(s) specified by DS_ATTRIBUTE_VALUES should be added to it.
  - DS_REMOVE_ATTRIBUTE

    The specified attribute is currently in the entry and should be deleted from the entry. Any values specified by DS_ATTRIBUTE_VALUES are ignored.
  - DS_REMOVE_VALUES

    The specified attribute is currently in the entry. One or more values, specified by DS_ATTRIBUTE_VALUES, should be removed from it.

Example Definition of a DS_C_ENTRY_MOD_LIST Object

The following code fragment is an example definition of a DS_C_ENTRY_MOD_LIST object.

```
OM_string my_uuid;

static OM_descriptor    Entry_Mod_Object[] = {
 OM_OID_DESC(OM_CLASS, DS_C_ENTRY_MOD),
 OM_OID_DESC(DS_ATTRIBUTE_TYPE, DSX_UUID),
 {DS_ATTRIBUTE_VALUES, OM_S_OCTET_STRING, my_uuid},
 {DS_MOD_TYPE, OM_S_ENUMERATION, DS_ADD_ATTRIBUTE},
 OM_NULL_DESCRIPTOR
};

static OM_descriptor    Entry_Mod_List_Object[] = {
 OM_OID_DESC(OM_CLASS, DS_C_ENTRY_MOD_LIST),
 {DS_CHANGES, OM_S_OBJECT, {0, Entry_Mod_Object}},
 OM_NULL_DESCRIPTOR
};
```

The DS_C_ENTRY_INFO_SELECTION Object

When you call **ds_read( )** to read one or more attributes from a CDS entry, you specify in the DS_C_ENTRY_INFO_SELECTION object the entry attributes you want to read.

The DS_C_ENTRY_INFO_SELECTION object contains the attributes shown in Figure 18 on page 73.

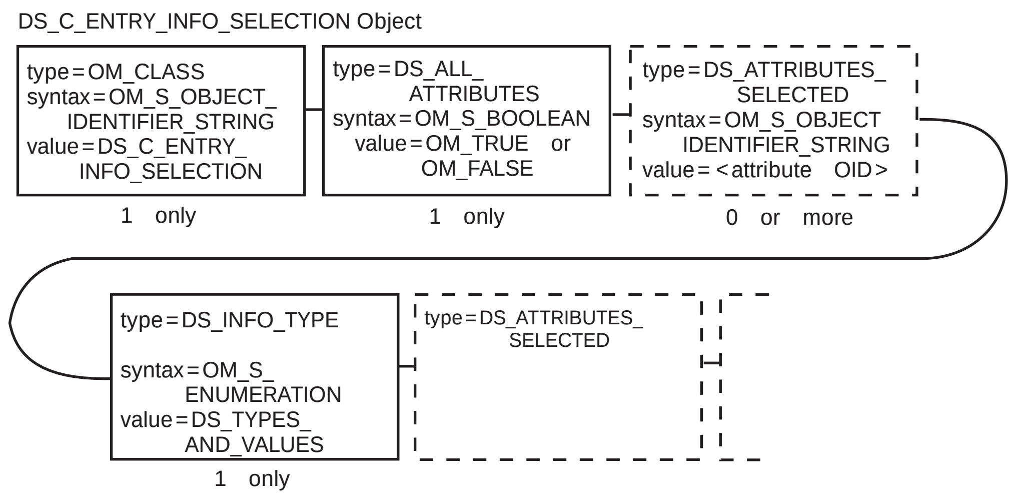

DS_C_ENTRY_INFO_SELECTION Object
type = OM_CLASS
syntax = OM_S_OBJECT_IDENTIFIER_STRING
value = DS_C_ENTRY_INFO_SELECTION
1 only
type = DS_ALL_ATTRIBUTES
syntax = OM_S_BOOLEAN
value = OM_TRUE or OM_FALSE
1 only
type = DS_ATTRIBUTES_SELECTED
syntax = OM_S_OBJECT IDENTIFIER_STRING
value = <attribute OID>
0 or more
type = DS_INFO_TYPE
syntax = OM_S_ENUMERATION
value = DS_TYPES_AND_VALUES
1 only
type = DS_ATTRIBUTES_SELECTED

Figure 18. The DS_C_ENTRY_INFO_SELECTION Object

Note that this object class has no subobjects.

- OM_CLASS

  The value of this attribute is an OID string that identifies the object's class; its value is always DS_C_ENTRY_INFO_SELECTION.

- DS_ALL_ATTRIBUTES

  This attribute is a simple Boolean option whose value indicates whether all the entry's attributes are to be read, or only some of them. Its possible values are as follows:

  – OM_TRUE, meaning that all attributes in the directory entry should be read. Any values specified by the DS_ATTRIBUTES_SELECTED attribute are ignored.

  – OM_FALSE, meaning that only some of the entry attributes should be read; namely, those specified by the DS_ATTRIBUTES_SELECTED attribute.

- DS_ATTRIBUTES_SELECTED

  The value of this attribute, which is an OID string, identifies the entry attribute to be read. Note that this attribute's value has meaning only if the value of DS_ALL_ATTRIBUTES is OM_FALSE; if it is OM_TRUE, the value of DS_ATTRIBUTES_SELECTED is ignored.

  Note also that there are multiple instances of this attribute if more than one attribute, but not all of them, is to be selected for reading. Each separate instance of DS_ATTRIBUTES_SELECTED has as its value an OID string that identifies one directory entry attribute to be read. If DS_ATTRIBUTES_SELECTED is present but does not have a value, **ds_read( )** reads the entry but does not return any attribute data; this technique can be used to verify the existence of a directory entry.

- DS_INFO_TYPE

  The value of this attribute specifies what information is to be read from each attribute specified by DS_ATTRIBUTES_SELECTED. The two possible values are as follows:

  – DS_TYPES_ONLY, meaning that only the attribute types of the selected attributes should be read.

  – DS_TYPES_AND_VALUES, meaning that both the attribute types and the attribute values of the selected attributes should be read.

Example Definition of a DS_C_ENTRY_INFO_SELECTION Object

The following code fragment provides an example definition of a DS_C_ENTRY_INFO_SELECTION object.

```
static OM_descriptor    Entry_Info_Select_Object[] = {
 OM_OID_DESC(OM_CLASS, DS_C_ENTRY_INFO_SELECTION),
 OM_OID_DESC(DS_ATTRIBUTES_SELECTED, DSX_A_CDS_Class),
 {DS_ALL_ATTRIBUTES, OM_S_BOOLEAN, OM_FALSE},
 {DS_INFO_TYPE, OM_S_ENUMERATION, DS_TYPES_AND_VALUES},
 OM_NULL_DESCRIPTOR
};
```

Attribute and Data Type Translation

This section provides translations between CDS and XDS for attributes and data types. Table 5 provides the OM syntax for CDS attributes. Table 6 provides the OM syntax for CDS data types. Table 7 on page 75 defines the mapping of CDS data types to OM syntaxes.

Table 5. CDS Attributes to OM Syntax Translation

CDS Attribute	**OM Syntax**
CDS_CTS	OM_S_OCTET_STRING
CDS_UTS	OM_S_OCTET_STRING
CDS_Class	OM_S_OCTET_STRING
CDS_ClassVersion	OM_S_INTEGER
CDS_ObjectUID	OM_S_OCTET_STRING
CDS_AllUpTo	OM_S_OCTET_STRING
CDS_Convergence	OM_S_INTEGER
CDS_InCHName	OM_S_INTEGER
CDS_DirectoryVersion	OM_S_INTEGER
CDS_UpgradeTo	OM_S_INTEGER
CDS_LinkTimeout	OM_S_INTEGER
CDS_Towers	OM_S_OCTET_STRING

Table 6. OM Syntax to CDS Data Types Translation

OM Syntax	**CDS Data Type**
OM_S_TELETEX_STRING	cds_char
OM_S_OBJECT_IDENTIFIER_STRING	cds_byte
OM_S_OCTET_STRING	cds_byte
OM_S_PRINTABLE_STRING	cds_char
OM_S_NUMERIC_STRING	cds_char
OM_S_BOOLEAN	cds_long
OM_S_INTEGER	cds_long
OM_S_UTC_TIME_STRING	cds_char
OM_S_ENCODING_STRING	cds_byte

Table 7. CDS Data Types to OM Syntax Translation

CDS Data Type	**OM Syntax**
cds_none	OM_S_NULL
cds_long	OM_S_INTEGER
cds_short	OM_S_INTEGER
cds_small	OM_S_INTEGER
cds_uuid	OM_S_OCTET_STRING
cds_Timestamp	OM_S_OCTET_STRING
cds_Version	OM_S_PRINTABLE_STRING
cds_char	OM_S_TELETEX_STRING
cds_byte	OM_S_OCTET_STRING

Part 3. XDS/XOM Application Programming

This section is an overview of programming using XDS/XOM.

“Chapter 5. XOM Programming” on page 83 describes XOM programming, and “Chapter 6. XDS Programming” on page 123 describes XDS programming. “Chapter 7. Using Threads With The XDS/XOM API” on page 145 describes how to use threads with XDS and XOM, and “Chapter 8. XDS/XOM Convenience Routines” on page 153 describes the XDS and XOM convenience routines.

Chapter 4. XDS API Logging

The XDS API logging facility displays informational and error messages for XDS functions. In addition, the input and output arguments to XDS function calls can also be displayed. For each XDS object, its OM types, syntaxes, and values are displayed recursively. A number of different display formats can be selected for the XDS objects. These are selected by setting the value of the environment variable `XDS_LOG` as shown in Table 8.

Logging can be activated dynamically at run-time by setting the environment variable `XDS_LOG`.

Table 8. XDS_LOG Values

XDS_LOG Values	Result	Example
Bit 1 = on	Display arguments, messages, results, and errors	N/A
Bit 1 = off	Display messages only (all other bits ignored)	N/A
Bit 2 = on	Display result and error objects as private objects	N/A
Bit 2 = off	Display result and error objects as public objects	N/A
Bit 3 = on	Object identifiers displayed as specified in fourth bit	N/A
Bit 3 = off	Object identifiers displayed as symbolic constants	**DS_C_SESSION**
Bit 4 = on	Object identifiers displayed as dotted-decimal	2.5.4.35
Bit 4 = off	Object identifiers displayed as hexadecimal bytes	`\x55\x04\x23`
Bit 5 = on	Syntaxes displayed as integers	127
Bit 5 = off	Syntaxes displayed as symbolic constants	**OM_S_OBJECT**

The bits shown in the previous table can be combined. For example, the following command sequence sets **XDS_LOG** to 5 (`0101` in binary):

```
XDS_LOG=5; export XDS_LOG
```

In the previous example, the logging facility is directed to display arguments, messages, results, and errors, to convert results and errors into public objects (for display purposes only) and to display object identifiers as hexadecimal bytes. It also displays OM syntaxes as symbolic constants. OM types are always displayed as integers. Normally, **XDS_LOG** should be set to 0. If full tracing is required, set **XDS_LOG** to 1.

The location of the log file can be set by means of the **D2_LOG_DIR** environment variable. For example, the following places the log files in **/tmp/log**.

```
D2_LOG_DIR=/tmp/log; export D2_LOG_DIR
```

A separate log file is created for each process that uses the XDS API. The log file names have the following format:

```
log_xds.pid
```

where pid is the process ID of the application. If the environment variable **D2_LOG_DIR** is not set, the files are placed in the directory set by the **HOME** environment variable.

Logging Format

The following general display format is used by the logging facility:

```
identifier-name = {
  { type, syntax, value },
  { type, syntax, value },
  :
  :
  etc. }; /* identifier-name */
```

where:

- **type** -- The integer defined for the specified type
- **syntax** -- A symbolic constant for the specified syntax. A **+L** is appended to the syntax label if the **OM_S_LOCAL_STRING** bit is set in the **OM_syntax** field.
- **value**
  - An integer (if **syntax** is **OM_S_INTEGER** or **OM_S_ENUMERATION**)
  - **OM_FALSE** or **OM_TRUE** (if **syntax** is **OM_S_BOOLEAN**)
  - Symbolic constant, dotted-decimal notation, or hexadecimal bytes (if **syntax** is **OM_S_OBJECT_ID_STRING**)
  - Quoted-string (if **syntax** is any other type of string)
  - Another object (if **synta**x is **OM_S_OBJECT**)

Note: The terminating `NULL` descriptor is expected but not displayed.

Examples

These examples show how a selection of XDS objects are displayed by the logging facility.

The following filter selects entries that do not have the value secret for the **DS_A_USER_PASSWORD** attribute. The **DS_FILTER_TYPE** has the value **DS_NOT**. It contains a single **DS_C_FILTER_ITEM** attribute. **DS_C_FILTER_ITEM** tests for equality against the **DS_A_USER_PASSWORD** attribute.

```
my_filter = {
   { OM_CLASS, OM_S_OBJECT_ID_STRING,    DS_C_FILTER },
   { DS_FILTER_ITEMS, OM_S_OBJECT,
     {
       { OM_CLASS, OM_S_OBJECT_ID_STRING, DS_C_FILTER_ITEM },
       { DS_FILTER_ITEM_TYPE, OM_S_ENUMERATION, 0 },
       { DS_ATTRIBUTE_TYPE, OM_S_OBJECT_ID_STRING, DS_A_USER_PASSWORD },
       { DS_ATTRIBUTE_VALUES, OM_S_OCTET_STRING, "secret" },
     }
   }
   { DS_FILTER_TYPE, OM_S_ENUMERATION, 3 },
}; /* my_filter */
```

The following example shows logging output if the interface logger encounters a `NULL` pointer. The `NULL` pointer is flagged as follows:

```
my_session = {
   { OM_CLASS, OM_S_OBJECT_ID_STRING, DS_C_SESSION },
   { DS_DSA_NAME, OM_S_OBJECT, ---WARNING: NULL pointer encountered--- },
}; /* my_session */
```

The following example shows logging output if the interface logger encounters a private object. The private object is displayed as follows:

```
bound_session = {
   { OM_PRIVATE_OBJECT,    OM_S_OBJECT_ID_STRING, DS_C_SESSION } ...
}; /* bound_session */
```

The following example shows how a five-part DSA distinguished name is displayed (`/C=de/O=sni/OU=ap/CN=dsa/CN=dsa-ml`):

```
dsa_name = {
   { DS_DSA_NAME, OM_S_OBJECT,
     {
       { OM_CLASS, OM_S_OBJECT_ID_STRING, DS_C_DS_DN },
       { DS_RDNS, OM_S_OBJECT,
         {
           { OM_CLASS, OM_S_OBJECT_ID_STRING, DS_C_DS_RDN },
             { DS_AVAS, OM_S_OBJECT,
               {
                 {OM_CLASS, OM_S_OBJECT_ID_STRING, DS_C_AVA },
                 { DS_ATTRIBUTE_TYPE, OM_S_OBJECT_ID_STRING, DS_A_COUNTRY_NAME },
                 { DS_ATTRIBUTE_VALUES, OM_S_PRINTABLE_STRING, "de" },
               }
             }
           }
         }
         { DS_RDNS, OM_S_OBJECT,
           {
             {OM_CLASS, OM_S_OBJECT_ID_STRING, DS_C_DS_RDN },
             { DS_AVAS, OM_S_OBJECT,
               {
                 {OM_CLASS, OM_S_OBJECT_ID_STRING, DS_C_AVA },
                 { DS_ATTRIBUTE_TYPE, OM_S_OBJECT_ID_STRING, DS_A_ORG_NAME },
                 { DS_ATTRIBUTE_VALUES, OM_S_TELETEX_STRING, "sni" },
               }
             }
           }
         }
         { DS_RDNS, OM_S_OBJECT,
           {
             {OM_CLASS, OM_S_OBJECT_ID_STRING, DS_C_DS_RDN },
             { DS_AVAS, OM_S_OBJECT,
               {
                 { OM_CLASS, OM_S_OBJECT_ID_STRING, DS_C_AVA },
                 { DS_ATTRIBUTE_TYPE, OM_S_OBJECT_ID_STRING, DS_A_ORG_UNIT_NAME },
                 { DS_ATTRIBUTE_VALUES, OM_S_TELETEX_STRING, "ap" },
               }
             }
           }
         }
         { DS_RDNS, OM_S_OBJECT,
           {
             { OM_CLASS, OM_S_OBJECT_ID_STRING, DS_C_DS_RDN },
             { DS_AVAS, OM_S_OBJECT,
               {
                 { OM_CLASS, OM_S_OBJECT_ID_STRING, DS_C_AVA },
                 { DS_ATTRIBUTE, OM_S_OBJECT_ID_STRING, DS_A_COMMON_NAME },
                 { DS_ATTRIBUTE_VALUES, OM_S_TELETEX_STRING, "dsa" },
               }
             }
           }
         }
         { DS_RDNS, OM_S_OBJECT,
           {
             { OM_CLASS, OM_S_OBJECT_ID_STRING, DS_C_DS_RDN },
             { DS_AVAS, OM_S_OBJECT,
               {
                 { OM_CLASS, OM_S_OBJECT_ID_STRING, DS_C_AVA },
                 { DS_ATTRIBUTE_TYPE, OM_S_OBJECT_ID_STRING, DS_A_COMMON_NAME },
                 { DS_ATTRIBUTE_VALUES, OM_S_TELETEX_STRING, "dsa-m1" },
               }
             }
           }
         }
       }
     }
 };  /* dsa_name */
```

Chapter 5. XOM Programming

XOM API defines a general-purpose interface for use in conjunction with other application-specific APIs for OSI services, such as XDS API to CDS directory services. It presents the application programmer with a uniform information architecture based on the concept of groups, classes, and similar information objects.

This chapter describes some of the basic concepts required to understand and use the XOM API effectively.

The following names refer to the complete XDS example programs, that are located in **/opt/dcelocal/examples/xdsxom**

- **add_list.c (add_list.h)**
- **cds_xmpl.c (cds_xmpl.h)**
- **example.c (example.h)**
- **thradd.c (thradd.h)**

For multithreaded XDS/XOM applications, please refer to “Chapter 7. Using Threads With The XDS/XOM API” on page 145. For use of the XDS/XOM convenience functions, please refer to “Chapter 8. XDS/XOM Convenience Routines” on page 153.

OM Objects

The purpose of XOM API is to provide an interface to manage complex information objects. These information objects belong to classes and have attributes associated with them. There are two distinct kinds of classes and attributes that are used throughout the directory service documentation: *directory* classes and attributes and *OM* classes and attributes.

The directory classes and attributes defined for XDS API correspond to entries that make up the objects in the directory. These classes and attributes are defined in the X.500 directory standard and by additional GDS extensions created for DCE. Other APIs, such as the X.400 API, which is the application interface for the industry standard X.400 electronic mail service, define their own set of objects in terms of classes and attributes. OM classes and OM attributes are used to model the objects in the directory.

XOM API provides a common information architecture so that the information objects defined for any API that conforms to this architectural model can be shared. Different application service interfaces can communicate by using this common way of defining objects by means of workspaces. A workspace is simply a common work area where objects defined by a service can be accessed and manipulated. In turn, XOM API provides a set of standard functions that perform common operations on these objects in a workspace. Two different APIs can share information by copying data from one workspace to another.

OM Object Attributes

OM objects are composed of OM attributes. OM objects may contain zero or more OM attributes. Every OM attribute has zero or more values. An attribute comprises an integer that indicates the attribute's value. Each value is accompanied by an integer that indicates that value's syntax.

An OM attribute type is a category into which all the values of an OM attribute are placed on the basis of its purpose. Some OM attributes may either have zero, one, or multiple values. The OM attribute type is used as the name of the OM attribute.

A syntax is a category into which a value is placed on the basis of its form. `OM_S_PRINTABLE_STRING` is an example of a syntax.

An OM attribute value is an information item that can be viewed as a characteristic or property of the OM object of which it is a part.

OM attribute types and syntaxes have integer values and symbolic equivalents assigned to them for ease of use by naming authorities in the various API specifications. The integers that are assigned to the OM attribute type and syntax are fixed, but the attribute values may change. These OM attribute types and syntaxes are defined in the DCE implementation of XDS and XOM APIs in header files that are included with the software along with additional OM attributes specific to the DCE implementation.

Figure 19 shows the internal structure of an OM object.

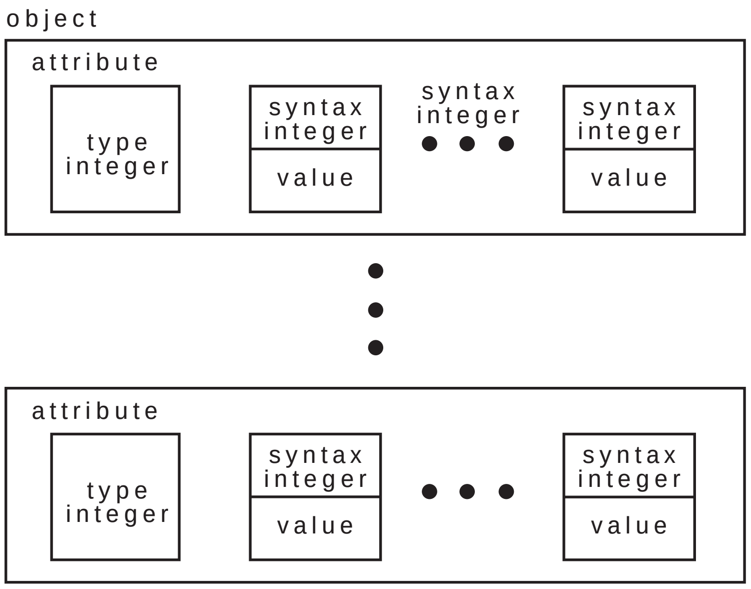


Figure 19. The Internal Structure of an OM Object

For example, the tables in Figure 20 on page 86 show the OM attributes, syntax, and values for the OM class `DS_C_ENTRY_INFO_SELECTION`, and how the integer values are mapped to corresponding names in the **xom.h** and **xds.h** header files. Refer to "Chapter 10. XDS Class Definitions" on page 189 for a complete description of `DS_C_ENTRY_INFO_SELECTION` and the accompanying table.

DS_C_ENTRY_INFO_SELECTION is a subclass of OM_C_OBJECT. This information is supplied in the description of this OM class in “Chapter 14. Object Management Package” on page 255. As such, DS_C_ENTRY_INFO_SELECTION inherits the OM attributes of OM_C_OBJECT. The only OM attribute of OM_C_OBJECT is OM_CLASS. OM_CLASS identifies the object’s OM class, which in this case is DS_C_ENTRY_INFO_SELECTION. DS_C_ENTRY_INFO_SELECTION identifies information to be extracted from a directory entry and has the following OM attributes, in addition to those inherited from OM_C_OBJECT:

- DS_ALL_ATTRIBUTES
- DS_ATTRIBUTES_SELECTED
- DS_INFO_TYPE

As part of an XDS function call, DS_ALL_ATTRIBUTES specifies to the directory service whether all the attributes of a directory entry are relevant to the application program. It can take the values OM_TRUE or OM_FALSE. These values are defined to be of syntax OM_S_BOOLEAN. The value OM_TRUE indicates that information is requested on all attributes in the directory entry. The value OM_FALSE indicates that information is only requested on those attributes that are listed in the OM attribute DS_ATTRIBUTES_SELECTED.

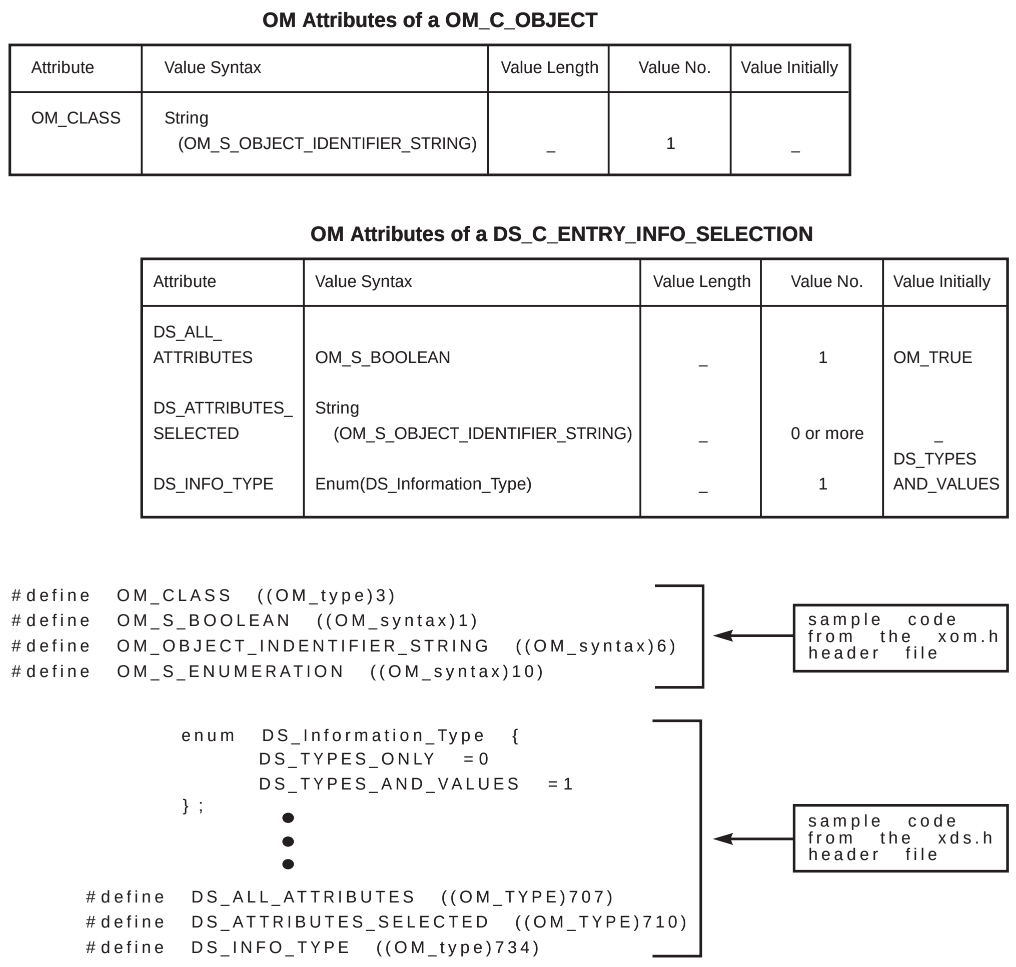
DS_ATTRIBUTES_SELECTED lists the types of attributes in the entry from which information is to be extracted. The syntax of the value is specified as OM_S_OBJECT_IDENTIFIER_STRING.

OM_S_OBJECT _IDENTIFIER_STRING contains an octet string of integers that are BER encoded object identifiers of the types of OM attributes in the OM attribute list. The value of DS_ATTRIBUTES _SELECTED is significant only if the value of DS_ALL_ATTRIBUTES is OM_FALSE, as described previously.

DS_INFO_TYPE identifies the information that is to be extracted from each OM attribute identified. The syntax of the value is specified as Enum(DS_Information_Type). DS_INFO_TYPE is an enumerated type that has two possible values: DS_TYPES_ONLY and DS_TYPES_ AND_VALUES. DS_TYPES_ONLY indicates that only the attribute types in the entry are returned by the directory service operation. DS_TYPES_AND_VALUES indicates that both the types and the values of the attributes in the directory entry are returned.

A typical directory service operation, such as a read operation (**ds_read( )**), requires the *entry_information_selection* parameter to specify to the directory service the information to be extracted from the directory entry. This *entry_information_selection* parameter is built by the application program as a public object (“Public Objects” on page 89 describes how to create a public object), and is included as a parameter to the **ds_read( )** function call, as shown in the following code fragment from example.c:

```
/*
 * Public Object ("Descriptor List") for
 * Entry-Information-Selection
 * parameter to ds_read().
 */
 OM_descriptor selection[] = {
 OM_OID_DESC(OM_CLASS,DS_C_ENTRY_INFO_SELECTION),
 { DS_ALL_ATTRIBUTES, OM_S_BOOLEAN, { OM_FALSE, NULL } },
 OM_OID_DESC(DS_ATTRIBUTES_SELECTED, DS_A_PHONE_NBR),
 { DS_INFO_TYPE,OM_S_ENUMERATION,
 { DS_TYPES_AND_VALUES,NULL } },
 OM_NULL_DESCRIPTOR
 };

 CHECK_DS_CALL(ds_read(session, DS_DEFAULT_CONTEXT,
                  name, selection, &result,
&invoke_id));
```



OM Attributes of a OM_C_OBJECT

Attribute	Value Syntax	Value Length	Value No.	Value Initially
OM_CLASS	String (OM_S_OBJECT_IDENTIFIER_STRING)	_	1	_

OM Attributes of a DS_C_ENTRY_INFO_SELECTION

Attribute	Value Syntax	Value Length	Value No.	Value Initially
DS_ALL_ ATTRIBUTES	OM_S_BOOLEAN	_	1	OM_TRUE
DS_ATTRIBUTES_ SELECTED	String (OM_S_OBJECT_IDENTIFIER_STRING)	_	0 or more	_
DS_INFO_TYPE	Enum(DS_Information_Type)	_	1	DS_TYPES AND_VALUES

```
#define  OM_CLASS  ((OM_type)3)
#define  OM_S_BOOLEAN  ((OM_syntax)1)
#define  OM_OBJECT_INDENTIFIER_STRING  ((OM_syntax)6)
#define  OM_S_ENUMERATION  ((OM_syntax)10)
```

```
      enum  DS_Information_Type  {
            DS_TYPES_ONLY  =0
            DS_TYPES_AND_VALUES  =1
      };
             ●
             ●
             ●
#define  DS_ALL_ATTRIBUTES  ((OM_TYPE)707)
#define  DS_ATTRIBUTES_SELECTED  ((OM_TYPE)710)
#define  DS_INFO_TYPE  ((OM_type)734)
```

Figure 20. Mapping the Class Definition of DS_C_ENTRY_INFO_SELECTION

Object Identifiers

OM classes are uniquely identifiable by means of ASN.1 object identifiers. OM classes have mandatory and optional OM attributes. Each OM attribute has a type, value, and syntax. OM objects are instances of OM classes that are uniquely identifiable by means of ASN.1 object identifiers. The syntax of values defined for these OM object classes and OM attributes are representations at a higher level of abstraction so that implementors can provide the necessary high-level language binding for their own implementations of the various application interfaces, such as XDS API.

The DCE implementation uses C language to define the internal representation of OM classes and OM attributes. These definitions are supplied in the header files that are included as part of XDS and the XOM API.

OM classes are defined as symbolic constants that correspond to ASN.1 object identifiers. An ASN.1 object identifier is a sequence of integers that uniquely identifies a specific class. OM attribute type and syntax are defined as integer constants. These standardized definitions provide application programs with a uniform and stable naming environment in which to perform directory operations. Registration authorities are responsible for allocating the unique object identifiers.

The following code fragment from the **xdsbdcp.h** (the basic directory contents package) header file contains the symbolic constant `OMP_O_DS_A_COUNTRY_NAME`:

```
#ifndef dsP_attributeType /*
joint-iso-ccitt(2) ds(5) attributeType(4)*/
#define dsP_attributeType(X) ("\x55\x04" #X)
#endif

#define OMP_O_DS_A_COUNTRY_NAME
dsp_attributeType(\x06)
```

It resolves to `2.5.4.6`, which is the object identifier value for the `Country-Name` attribute type as defined in the directory standard. The symbolic constant for the directory object class `Country` resolves to `2.5.6.2`, the corresponding object identifier in the directory standard. OM classes are defined in the header files in the same manner.

Note: 2.5.4.6 and 2.5.6.2 are object identifiers defined by the standards, not the BER encoding found in the header file which are `\x55\x04\x06` and `\x55\x06\x02`.

C Naming Conventions

In the DCE implementation of XDS and XOM APIs, all object identifiers start with the letters `ds`, `DS`, `MH`, or `OMP`. Note that the interface reserves *all* identifiers starting with the letters `dsP` and `omP` for internal use by implementations of the interface. It also reserves all identifiers starting with the letters `dsX`, `DSX`, `omX`, and `OMX` for vendor-specific extensions of the interface. Applications programmers should not use any identifier starting with these letters.

The C identifiers for interface elements are formed by using the following conventions:

- XDS API function names are specified entirely in lowercase letters and are prefixed by `ds_` (for example, **ds_read( )**).
- XOM API function names are specified entirely in lowercase letters and are prefixed by `om_` (for example, **om_get( )**).
- C function parameters are derived from the parameter and result names and are specified entirely in lowercase letters. In addition, the names of results have `_return` added as a suffix (for example, `operation_status_return`).
- OM class names are specified entirely in uppercase letters and are prefixed by `DS_C_` and `MH_C_` (for example, `DS_C_AVA`).
- OM attribute names are specified entirely in uppercase letters and are prefixed by `DS_` and `MH_` (for example, `DS_RDNS`).
- OM syntax names are specified entirely in uppercase letters and are prefixed by `OM_S_` (for example, `OM_S_PRINTABLE_STRING`).
- Directory class names are specified entirely in uppercase letters and are prefixed by `DS_O` (for example, `DS_O_ORG_PERSON`).

- Directory attribute names are specified entirely in uppercase letters and are prefixed by DS_A (for example, DS_A_COUNTRY_NAME).
- Errors are treated as a special case. Constants that are the possible values of the OM attribute DS_PROBLEM of a subclass of the OM class DS_C_ERROR are specified entirely in uppercase letters and are prefixed by DS_E_ (for example, DS_E_BAD_CLASS).
- The constants in the Value Length and Value Number columns of the OM class definition tables are also assigned identifiers. Where the upper limit in one of these columns is *not* 1, it is given a name that consists of the OM attribute name, prefixed by DS_VL_ for value length, or DS_VN_ for value number.
- The sequence of octets for each object identifier is also assigned an identifier for internal use by certain OM macros. These identifiers are all uppercase letters and are prefixed by OMP_O_.

Table 9 and Table 10 summarize the XDS and XOM naming conventions.

Table 9. C Naming Conventions for XDS

Item	**Prefix**
Reserved for implementors	dsP
Reserved for interface extensions	dsX
Reserved for interface extensions	DSX
XDS functions	ds_
Error problem values	DS_E_
OM class names	DS_C_, MH_C_
OM attribute names	DS_, MH_
OM value length limits	DS_VL_
OM value number limits	DS_VN_
Other constants	DS_, MH_
Attribute type	DS_A_
Object class	DS_O_

Table 10. C Naming Conventions for XOM

Element Type	**Prefix**
Data type	OM_
Data value	OM_
Data value (class)	OM_C_
Data value (syntax)	OM_S_
Data value component (structure member)	None
Function	om_
Function parameter	None
Function result	None
Macro	OM_
Reserved for use by implementors	OMP
Reserved for use by implementors	omP
Reserved for proprietary extension	omX
Reserved for proprietary extension	OMX

Public Objects

The ultimate aim of an application program is access to the directory to perform some operation on the contents of the directory. A user may request the telephone number or electronic mail address of a fellow employee. In order to access this information, the application performs a read operation on the directory so that information is extracted about a target object in the directory and manipulated locally within the application.

XDS functions that perform directory operations, such as **ds_read( )**, require *public* and *private* objects as input parameters. Typically, a public object is generated by an application program and contains the information required to access a target directory object. This information includes the AVAs and RDNs that make up a distinguished name of an entry in the directory. However, an application program may also generate a private object. Private objects are described in “Private Objects” on page 98.

A public object is created by using OM classes and OM attributes. These OM classes and OM attributes model the target object entry in the directory and provide other information required by the directory service to access the directory.

Descriptor Lists

A public object is represented by a sequence of `OM_descriptor` data structures that is built by the application program. A descriptor contains the type, syntax, and value for an OM attribute in a public object.

The data structure `OM_descriptor` is defined in the **xom.h** header file as follows:

```
typedef struct OM_descriptor_struct {
        OM_type                 type;
        OM_syntax               syntax;
        union OM_value_union    value;
}OM_descriptor;
```

Figure 21 on page 90 shows the representation of a public object in a descriptor list. The first descriptor in the list indicates the object’s OM class; the last descriptor is a NULL descriptor that signals the end of the list of OM attributes. In between the first and the last descriptor are the descriptors for the OM attributes of the object.

For example, the following represents the public object `country` in `example.c`:

```
static OM_descriptor        country[] = {
 OM_OID_DESC(OM_CLASS, DS_C_AVA),
 OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_COUNTRY_NAME),
 { DS_ATTRIBUTE_VALUES,OM_S_PRINTABLE_STRING,OM_STRING("US") },
 OM_NULL_DESCRIPTOR
 };
```

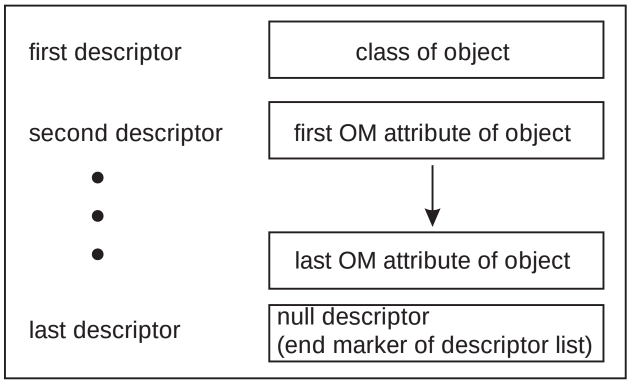

first descriptor
class of object
second descriptor
first OM attribute of object
last OM attribute of object
last descriptor
null descriptor
(end marker of descriptor list)

Figure 21. A Representation of a Public Object By Using a Descriptor List

The descriptor list is an array of data type `OM_descriptor` that defines the OM class, OM attribute types, syntax, and values that make up a public object.

The first descriptor gives the OM class of the object. The OM class of the object is defined by the OM attribute type, `OM_CLASS`. The `OM_OID_DESC` macro initializes the syntax and value of an object identifier, in this case to OM class `DS_C_AVA`, with the syntax of `OM_S_OBJECT_IDENTIFIER_STRING`. `OM_S_OBJECT_IDENTIFIER_STRING` is an OM syntax type that is assigned by definition in the macro to any OM attribute type and value parameters input to it.

The second descriptor defines the first OM attribute type, `DS_ATTRIBUTE_TYPE`, which has as its value `DS_A_COUNTRY_NAME` and syntax `OM_S_OBJECT_IDENTIFIER_STRING`.

The third descriptor specifies the AVA of an object entry in the directory. The `OM_OID_DESC` macro is not used here because `OM_OID_DESC` is only used to initialize values having `OM_S_OBJECT_IDENTIFIER_STRING` syntax. The OM attribute type is `DS_ATTRIBUTE_VALUES`, the syntax is `OM_S_PRINTABLE_STRING`, and the value is `US`. The `OM_STRING` macro creates a data value for a string data type (data type `OM_string`), in this case `OM_S_PRINTABLE_STRING`. A string is specified in terms of its length or whether or not it terminates with a NULL. (The `OM_STRING` macro is described in "The OM_STRING Macro" on page 121.)

The last descriptor is a NULL descriptor that marks the end of the public object definition. It is defined in the **xom.h** header file as follows:

```
#define OM_NULL_DESCRIPTOR
  { OM_NO_MORE_TYPES, OM_S_NO_MORE_SYNTAXES,
  { { 0, OM_ELEMENTS_UNSPECIFIED } } }
```

`OM_NULL_DESCRIPTOR` is OM attribute type `OM_NO_MORE_TYPES`, syntax `OM_S_NO_MORE_SYNTAXES`, and value `OM_ELEMENTS_UNSPECIFIED`.

Figure 22 on page 91 shows the composition of a descriptor list representing a public object.

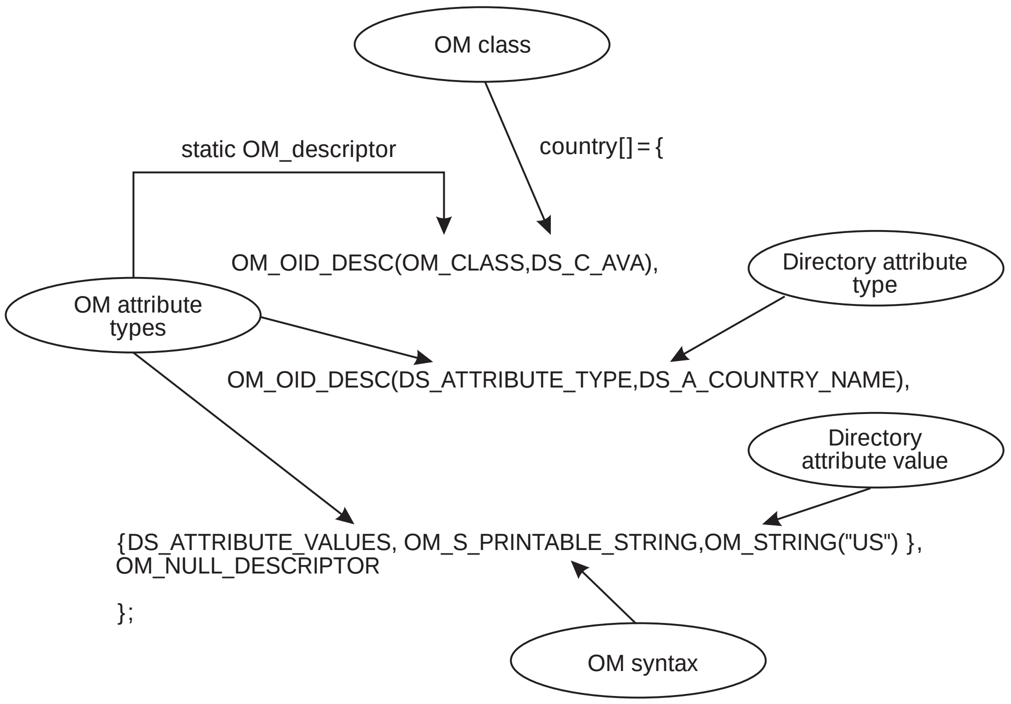

OM class
static OM_descriptor
country[] = {
OM_OID_DESC(OM_CLASS,DS_C_AVA),
Directory attribute type
OM attribute types
OM_OID_DESC(DS_ATTRIBUTE_TYPE,DS_A_COUNTRY_NAME),
Directory attribute value
{DS_ATTRIBUTE_VALUES, OM_S_PRINTABLE_STRING,OM_STRING("US") },
OM_NULL_DESCRIPTOR
};
OM syntax

Figure 22. A Descriptor List for the Public Object: country

Building the Distinguished Name as a Public Object

Recall that RDNs are built from AVAs, and a distinguished name is built from a series of RDNs. In a typical application program, several AVAs are defined in descriptor lists as public objects. These public objects are incorporated into descriptor lists that represent corresponding RDNs. Finally, the RDNs are incorporated into one descriptor list that represents the distinguished name of an object in the directory, as shown in Figure 23 on page 92. This descriptor list is included as one of the input parameters to a directory service function.

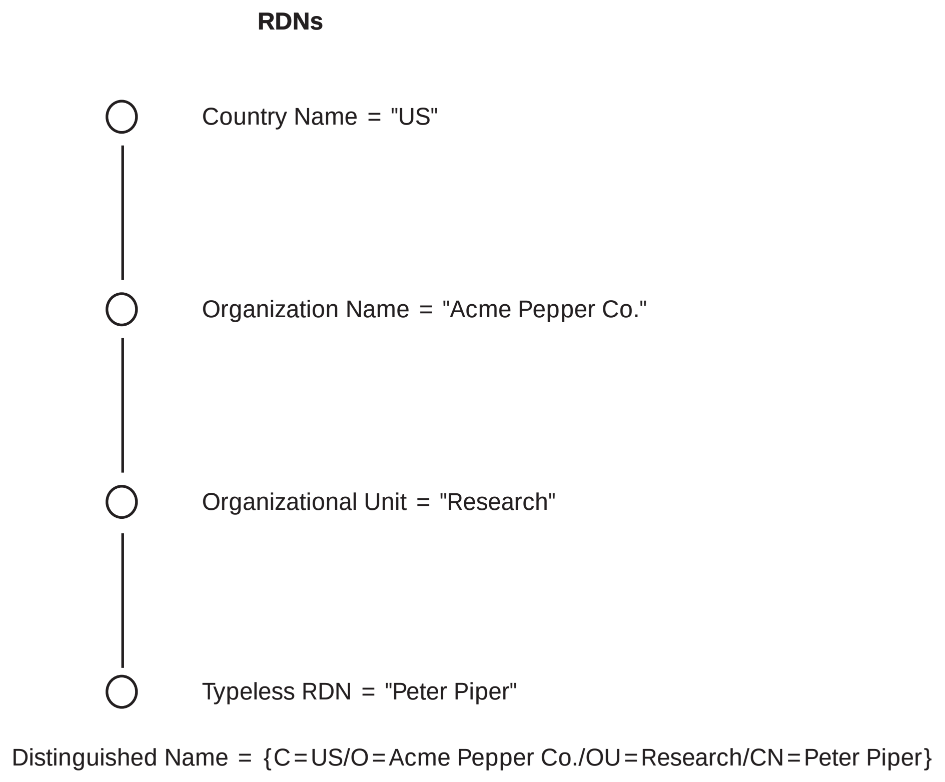

RDNs
Country Name = "US"
Organization Name = "Acme Pepper Co."
Organizational Unit = "Research"
Typeless RDN = "Peter Piper"
Distinguished Name = {C=US/O=Acme Pepper Co./OU=Research/CN=Peter Piper}

Figure 23. The Distinguished Name of "Peter Piper" in the DIT

The following code fragment from `example.c` shows how a distinguished name is built as a public object. The public object is the *name* parameter for a subsequent read operation call to the directory. The representation of a distinguished name in the DIT is shown in Figure 23.

The first section of code defines the four AVAs. These AVAs make the assertion to the directory service that the attribute values in the distinguished name of `Peter Piper` are valid and can therefore be read from the directory. The country name is `US`, the organization name is `Acme Pepper Co`, the organizational unit name is `Research`, and the common name is `Peter Piper`.

```
/*
 * Public Object ("Descriptor List") for Name parameter to
 * ds_read().
 * Build the Distinguished-Name of Peter Piper
 */

static OM_descriptor        country[] = {
 OM_OID_DESC(OM_CLASS, DS_C_AVA),
 OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_COUNTRY_NAME),
 { DS_ATTRIBUTE_VALUES,OM_S_PRINTABLE_STRING,OM_STRING("US") },
 OM_NULL_DESCRIPTOR
 };
static OM_descriptor        organization[] = {
 OM_OID_DESC(OM_CLASS, DS_C_AVA),
 OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_ORG_NAME),
 { DS_ATTRIBUTE_VALUES,OM_S_TELETEX_STRING,
OM_STRING("Acme Pepper Co") },
 OM_NULL_DESCRIPTOR
 };
static OM_descriptor        organizational_unit[] = {
 OM_OID_DESC(OM_CLASS, DS_C_AVA),
 OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_ORG_UNIT_NAME),
 { DS_ATTRIBUTE_VALUES,OM_S_TELETEX_STRING, OM_STRING("Research") },
 OM_NULL_DESCRIPTOR
 };
static OM_descriptor        common_name[] = {
 OM_OID_DESC(OM_CLASS, DS_C_AVA),
 OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_COMMON_NAME),
 { DS_ATTRIBUTE_VALUES,OM_S_TELETEX_STRING, OM_STRING("Peter Piper") },
 OM_NULL_DESCRIPTOR
 };
```

The next section of code is nested one level above the previously defined AVAs. Each RDN has a descriptor with OM attribute type `DS_AVAS` (indicating that it is OM attribute type `AVA`), a syntax of `OM_S_OBJECT`, and a value of the name of the descriptor array defined in the previous section of code for an AVA. The `rdn1` descriptor contains the descriptor list for the AVA country, the `rdn2` descriptor contains the descriptor list for the AVA organization, and so on.

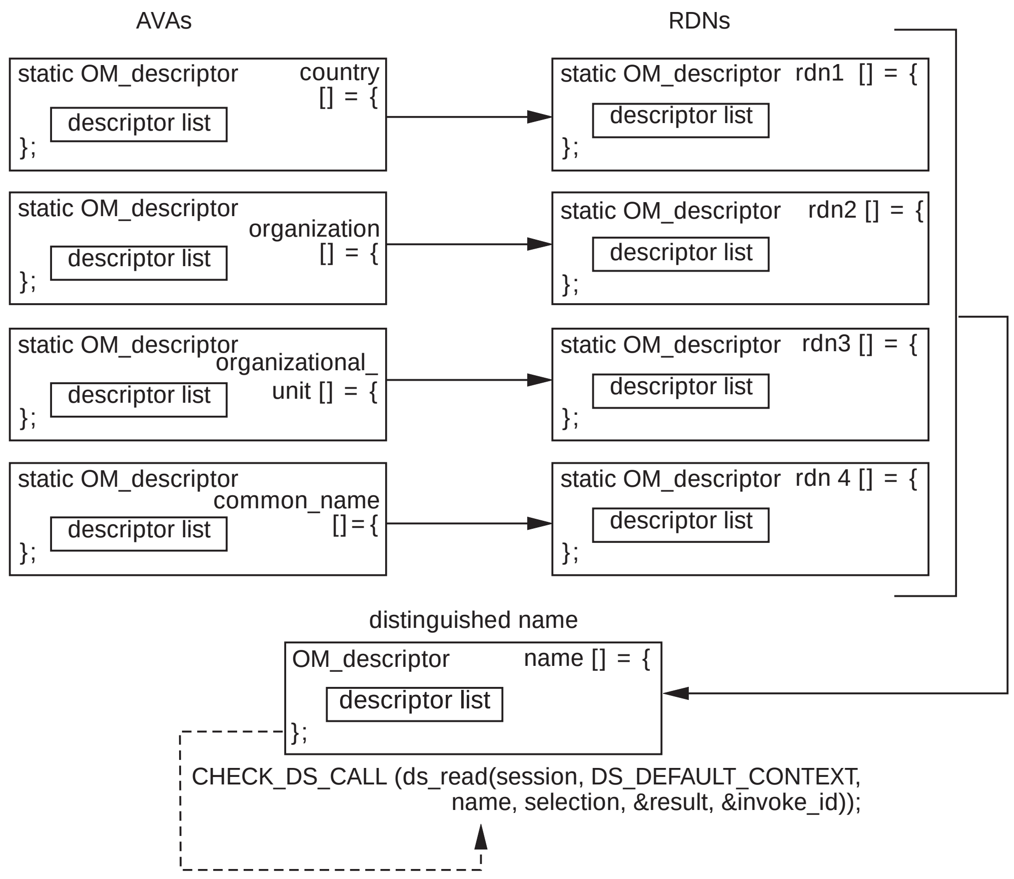
`OM_S_OBJECT` is a syntax that indicates that its value is a subobject. For example, the value for `DS_AVAS` is the previously defined object `country`. In this manner, a hierarchy of linked objects and subobjects can be constructed.

```
static OM_descriptor        rdn1[] = {
 OM_OID_DESC(OM_CLASS, DS_C_DS_RDN),
 { DS_AVAS, OM_S_OBJECT, { 0, country } },
 OM_NULL_DESCRIPTOR
 };
static OM_descriptor        rdn2[] = {
 OM_OID_DESC(OM_CLASS, DS_C_DS_RDN),
 { DS_AVAS, OM_S_OBJECT, { 0, organization } },
 OM_NULL_DESCRIPTOR
 };
static OM_descriptor        rdn3[] = {
 OM_OID_DESC(OM_CLASS, DS_C_DS_RDN),
 { DS_AVAS, OM_S_OBJECT, { 0, organizational_unit } },
 OM_NULL_DESCRIPTOR
  };
static OM_descriptor        rdn4[] = {
 OM_OID_DESC(OM_CLASS, DS_C_DS_RDN),
 { DS_AVAS, OM_S_OBJECT, { 0, common_name } },
 OM_NULL_DESCRIPTOR
 };
```

The next section of code contains the RDNs that make up the distinguished name, which is stored in the array of descriptors called *name*. It is made up of the OM class `DS_C_DS_DN` (representing a distinguished name) and four RDNs of OM attribute type `DS_RDNS` and syntax `OM_S_OBJECT`.

```
OM_descriptor       name[] = {
 OM_OID_DESC(OM_CLASS, DS_C_DS_DN),
 { DS_RDNS, OM_S_OBJECT, { 0, rdn1 } },
 { DS_RDNS, OM_S_OBJECT, { 0, rdn2 } },
 { DS_RDNS, OM_S_OBJECT, { 0, rdn3 } },
 { DS_RDNS, OM_S_OBJECT, { 0, rdn4 } },
 OM_NULL_DESCRIPTOR
 };
```

In summary, the distinguished name for `Peter Piper` is stored in the array of descriptors called *name*, which is composed of three nested levels of arrays of descriptors (see Figure 24 on page 94). The definitions for the AVAs are at the innermost level, the definitions for RDNs are at the next level up, and the distinguished name is at the top level.


AVAs
RDNs
static OM_descriptor country [] = {
descriptor list
};
static OM_descriptor organization [] = {
descriptor list
};
static OM_descriptor organizational_ unit [] = {
descriptor list
};
static OM_descriptor common_name []={
descriptor list
};
static OM_descriptor rdn1 [] = {
descriptor list
};
static OM_descriptor rdn2 [] = {
descriptor list
};
static OM_descriptor rdn3 [] = {
descriptor list
};
static OM_descriptor rdn 4 [] = {
descriptor list
};
distinguished name
OM_descriptor name [] = {
descriptor list
};

```
CHECK_DS_CALL (ds_read(session, DS_DEFAULT_CONTEXT,
                           name, selection, &result, &invoke_id));
```

Figure 24. Building a Distinguished Name

Figure 25 on page 95 shows a more general view of the structure distinguished name.

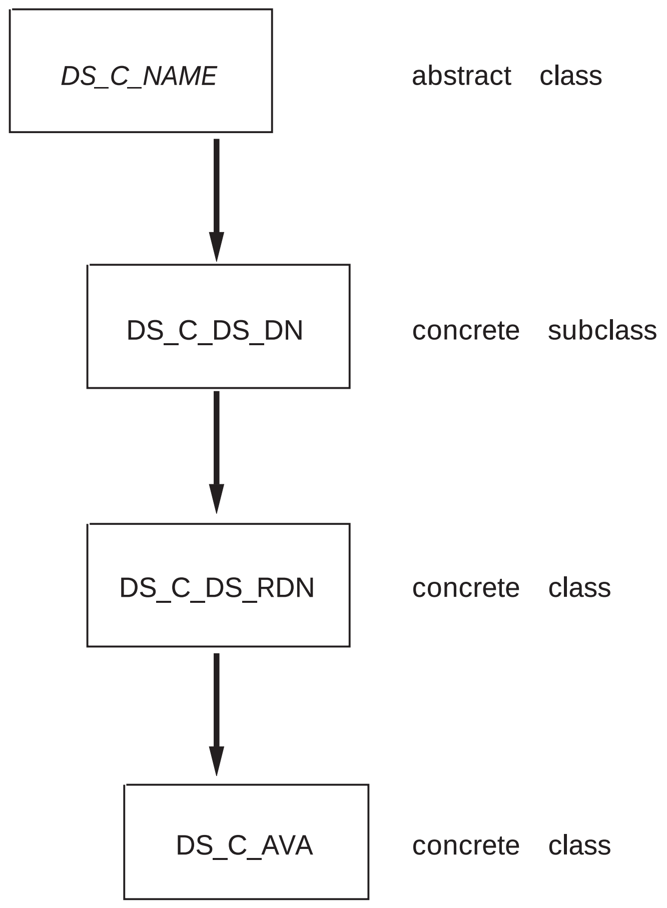


Figure 25. A Simplified View of the Structure of a Distinguished Name

The `name` descriptor defines a public object that is provided as the *name* parameter required by the XDS API read function call, **ds_read( )**, as follows. (XDS API function calls are described in detail in “Chapter 6. XDS Programming” on page 123.)

```
CHECK_DS_CALL(ds_read(session, DS_DEFAULT_CONTEXT,
                      name, selection, &result,
&invoke_id));
```

The result of the **ds_read( )** function call is in a private implementation-specific format; it is stored in a workspace and pointed to by `result`. The application program must use XOM function calls (described in “OM Function Calls” on page 114) to interpret the data and extract the information. This extraction process involves uncovering the nested data structures in a series of XOM function calls.

Client-Generated and Service-Generated Public Objects

There are two types of public objects: service-generated objects and client-generated objects. The distinguished name object described in the previous section is a client-generated public object because an application program (the client) created the data structure. As the creator of the public object, it is the responsibility of the application program to manage the memory resources allocated for it.

Service-generated public objects are created by the XOM service.
Service-generated public objects may be generated as a result of an XOM request. An XOM API function, such as **om_get( )**, converts a private object into a service-generated public object. This is necessary because XDS may return a pointer to data in private format that can only be interpreted by XOM functions such as **om_get( )**.

For example, Figure 26 on page 97 shows how the read request described in the previous example returns a pointer to an encoded data structure stored in `result`. This encoded data structure, referred to as a *private object* (described in the next section) is one of the input parameters to **om_get( )**. The **om_get( )** function provides a pointer to a public object (in this case, `entry`) as an output parameter. The public object is a data structure that has been interpreted by **om_get( )** and is accessible by the application program (the client). The information requested by the application in the read request is contained in the output parameter `entry`.

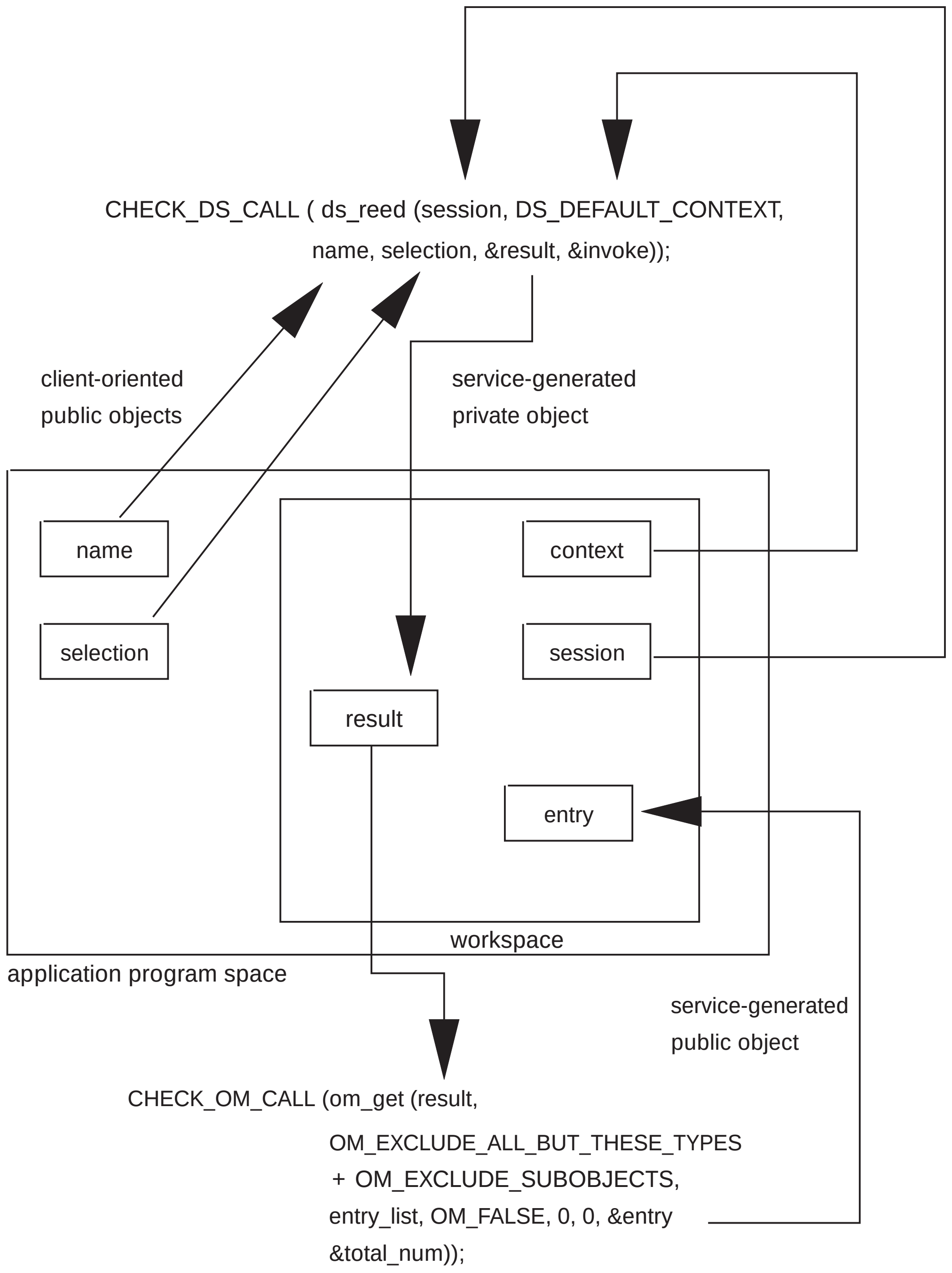

CHECK_DS_CALL (ds_reed (session, DS_DEFAULT_CONTEXT,
name, selection, &result, &invoke));
client-oriented
public objects
service-generated
private object
name
selection
context
session
result
entry
workspace
application program space
service-generated
public object
CHECK_OM_CALL (om_get (result,
OM_EXCLUDE_ALL_BUT_THESE_TYPES
+ OM_EXCLUDE_SUBOBJECTS,
entry_list, OM_FALSE, 0, 0, &entry
&total_num));

Figure 26. Client-Generated and Service-Generated Objects

The application program is responsible for managing the storage (memory) for the service-generated public object. This is an important point because it requires that the application issue a series of **om_delete( )** calls to delete the service-generated public object from memory. Because the data structures involved with directory service requests can be very large (often involving large subtrees of the DIT), it is imperative that the application programmer build into any application program the efficient management of memory resources.

The following code fragment from `example.h` demonstrates how storage for public and private objects is released by using a series of **om_delete( )** function calls after they are no longer needed by the application program. The data (a list of phone numbers associated with the name `Peter Piper` required by the application program) has already been extracted by using a series of **om_get( )** function calls, as follows:

```
/*  We can now safely release all the private objects
 *  and the public objects we no longer need
 */
CHECK_OM_CALL(om_delete(session));
CHECK_OM_CALL(om_delete(result));
CHECK_OM_CALL(om_delete(entry));
CHECK_OM_CALL(om_delete(attributes));
CHECK_DS_CALL(ds_shutdown(workspace));
```

Private Objects

Private objects are created dynamically by the service interface. In Figure 26 on page 97, the **ds_read( )** function returns a pointer to the data structure `result` in the workspace. This service-generated data structure is a private object in a private implementation-specific format, which requires a call to **om_get( )** to interpret the data. A private object is one of the required input parameters to XOM API functions (such as **om_get( )**), as shown in Figure 26 on page 97. Private objects are always service generated.

Table 11 compares private and public objects.

Table 11. Comparison of Private and Public Objects

Private	**Public**
Representation is implementation specific	Representation is defined in the API specification
Not directly accessible by the client	Directly accessible by the client
Manipulated by the client by using OM functions	Manipulated by the client by using programming constructs
Created in storage provided by the service	Is a service-generated object if created by the service Is a client-generated object if created by the client in storage provided by the client
Cannot be modified by the client directly, except through the service interface	If a client-generated object, can be modified directly by the client If a service-generated object, cannot be modified directly by the client, except through the service interface
Storage is allocated and released by the service	If a service-generated object, storage is allocated and released by the service If a client-generated object, storage is allocated and released by the client

Private objects can also be used as input to XOM and XDS API functions to improve program efficiency. For example, the output of a **ds_search( )** request can be used as input to **ds_read( )**. The search request returns the name of each entry in the search. If the application program requires the address and telephone number of each name, a **ds_read( )** operation can be performed on each name as a private object.

Object Classes

Objects are categorized into OM classes based on their purpose and internal structure. An object is an instance of its OM class. An OM class is characterized by OM attribute types that may appear in its instances. An OM class is uniquely identified by an ASN.1 object identifier.

Later in this section, it will be shown how OM classes are organized into groups of OM classes, called *packages*, that support some aspect of the directory service.

OM Class Hierarchy and Inheritance Properties

OM classes are related to each other in a tree hierarchy whose root is a special OM class called OM_C_OBJECT. Each of the other OM classes is the immediate subclass of precisely one other OM class. This tree structure is known as the *OM class hierarchy*. It is important because of the property of inheritance. The OM class hierarchy is defined by the XDS/XOM standards. DCE implements this hierarchy for XDS/XOM.

The OM attribute types that may exist in an instance of an OM class, but not in an instance of the OM class above it in the tree hierarchy, are said to be *specific* to that OM class. OM attributes that may appear in an object are those specific to its OM class as well as those inherited from OM classes above it in the tree. OM classes above an instance of an OM class in the tree are *superclasses* of that OM class. OM classes below an instance of an OM class are *subclasses* of that OM class.

For example, as shown in Figure 27, DS_C_ENTRY_INFO_SELECTION inherits its OM attributes from its superclass OM_C_OBJECT. The OM attributes DS_ALL_ATTRIBUTES, DS_ATTRIBUTES_SELECTED, and DS_INFO_TYPE are attributes specific to the OM class DS_C_ENTRY_INFO_SELECTION. The DS_C_ENTRY_INFO_SELECTION class has no subclasses.

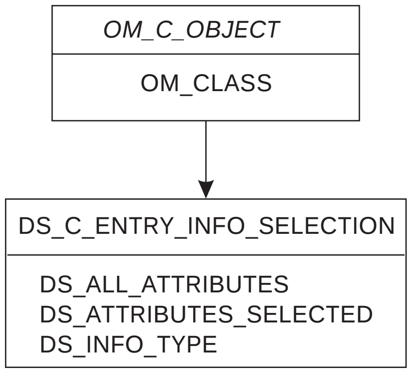


Figure 27. The OM Class DS_C_ENTRY_INFO_SELECTION

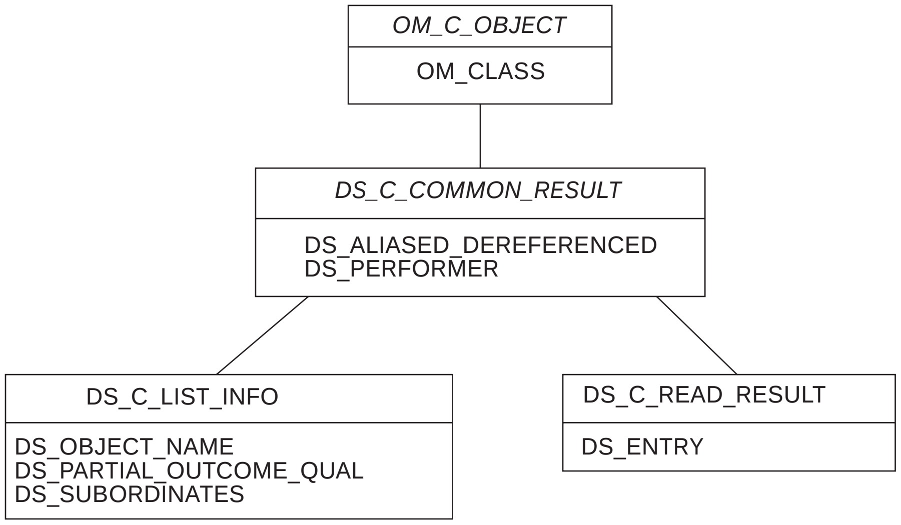
Another important point about OM class inheritance is that an instance of an OM class is also considered to be an instance of each of its superclasses and may appear wherever the interface requires an instance of any of those superclasses. For example, DS_C_DS_DN is a subclass of DS_C_NAME. Everywhere in an application program where DS_C_NAME is expected at the interface (as a parameter to **ds_read( )**, for example), it is permitted to supply DS_C_DS_DN.

Abstract and Concrete Classes

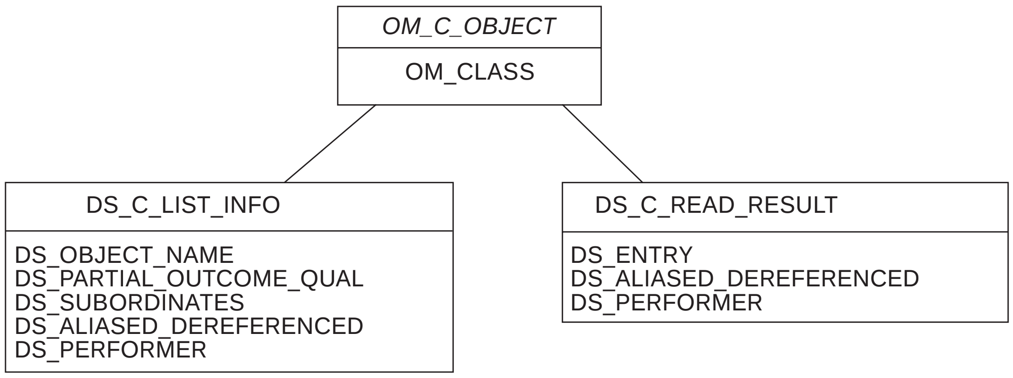
OM classes are defined as being either *abstract* or *concrete*. An abstract OM class is an OM class in which instances are not permitted. An abstract OM class may be defined so that subclasses can share a common set of OM attributes between them.

In contrast to abstract OM classes, instances of OM concrete classes are permitted. However, the definition of each OM concrete class may include the restriction that a client not be allowed to create instances of that OM class. For example, consider two alternative means of defining the OM classes used in XDS: `DS_C_LIST_INFO` and `DS_C_READ_RESULT`. `DS_C_LIST_INFO` and `DS_C_READ_RESULT` are subclasses of the abstract OM class `DS_C_COMMON_RESULT`.

Figure 28 on page 101 shows the relationship of `DS_C_LIST_INFO` and `DS_C_READ_RESULTS` when the abstract OM class `DS_C_COMMON_RESULT` is defined and when it is not defined. It demonstrates that the presence of an abstract OM class enables the programmer to develop applications that process information more efficiently.



DS_C_LIST_INFO and DS_C_READ_RESULT with the DS_C_COMMON_RESULT abstract class defined



DS_C_LIST_INFO and DS_C_READ_RESULT without the DS_C_COMMON_RESULT abstract class defined

Figure 28. Comparison of Two Classes With/Without an Abstract OM Class

The following list contains the hierarchy of concrete and abstract OM classes in the directory service package. Abstract OM classes are shown in italics. The indentation shows the class hierarchy; for example, the abstract class *OM_C_OBJECT* is a superclass of the abstract class *DS_C_COMMON_RESULTS*, which in turn is a superclass of the concrete class `DS_C_COMPARE_RESULT`.

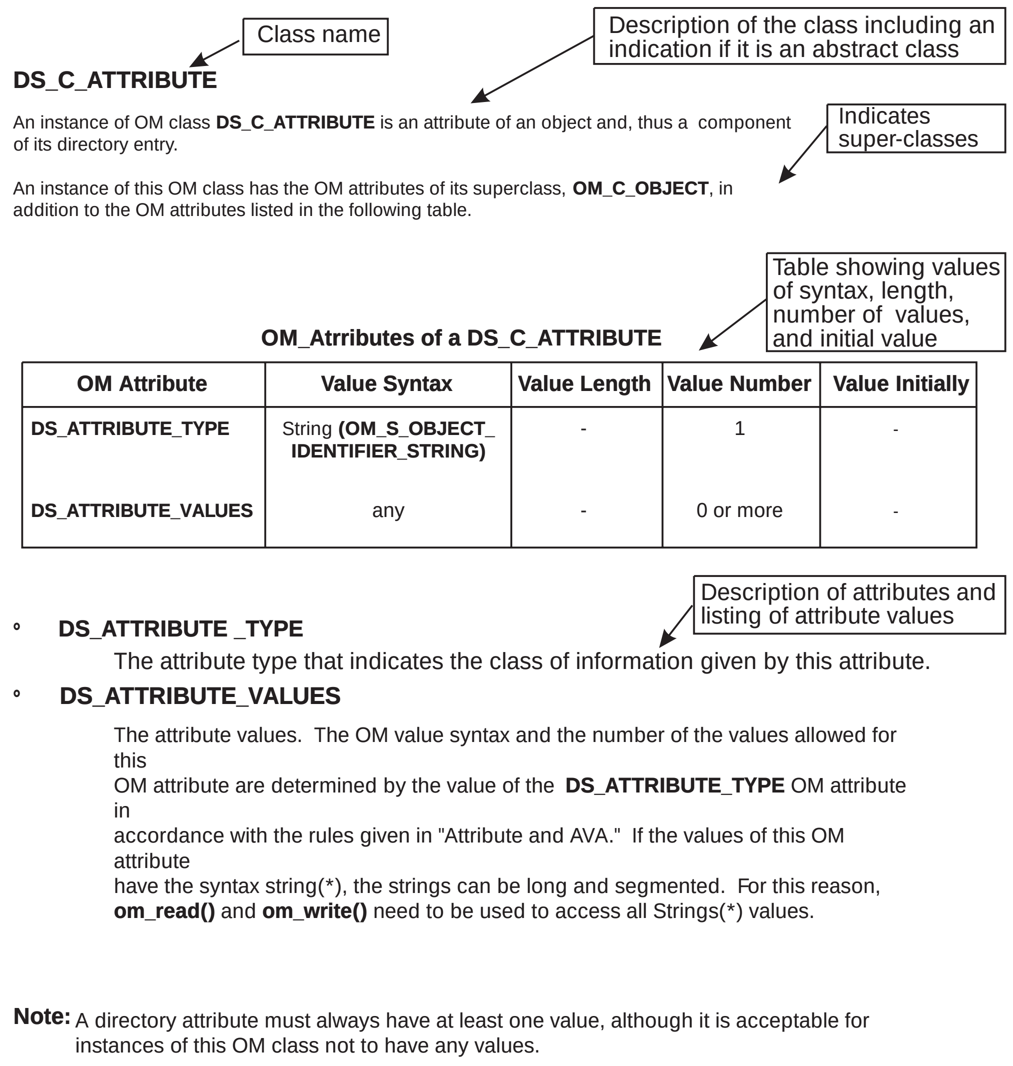
OM_C_OBJECT

- `DS_C_ACCESS_POINT`
- *DS_C_ADDRESS*
  - `DS_C_PRESENTATION_ADDRESS`
- `DS_C_ATTRIBUTE`
  - `DS_C_AVA`
  - `DS_C_ENTRY_MOD`
  - `DS_C_FILTER_ITEM`

- `DS_C_ATTRIBUTE_ERROR`
- `DS_C_ATTRIBUTE_LIST`
  - `DS_C_ENTRY_INFO`
- *DS_C_COMMON_RESULTS*
  - `DS_C_COMPARE_RESULT`
  - `DS_C_LIST_INFO`
  - `DS_C_READ_RESULT`
  - `DS_C_SEARCH_INFO`
- `DS_C_CONTEXT`
- `DS_C_CONTINUATION_REF`
  - `DS_C_REFERRAL`
- `DS_C_ENTRY_INFO_SELECTION`
- `DS_C_ENTRY_MOD_LIST`
- *DS_C_ERROR*
  - `DS_C_ABANDON_FAILED`
  - `DS_C_ATTRIBUTE_PROBLEM`
  - `DS_C_COMMUNICATIONS_ERROR`
  - `DS_C_LIBRARY_ERROR`
  - `DS_C_NAME_ERROR`
  - `DS_C_SECURITY_ERROR`
  - `DS_C_SERVICE_ERROR`
  - `DS_C_SYSTEM_ERROR`
  - `DS_C_UPDATE_ERROR`
- `DS_C_EXT`
- `DS_C_FILTER`
- `DS_C_LIST_INFO_ITEM`
- `DS_C_LIST_RESULT`
- *DS_C_NAME*
  - `DS_C_DS_DN`
- `DS_C_OPERATION_PROGRESS`
- `DS_C_PARTIAL_OUTCOME_QUAL`
- *DS_C_RELATIVE_NAME*
  - `DS_C_DS_RDN`
- `DS_C_SEARCH_RESULT`
- `DS_C_SESSION`

In summary, an OM class is defined with the following elements:
- OM class name (indicated by an object identifier)
- Identity of its immediate superclass
- Definitions of the OM attribute types specific to the OM class
- Indication whether the OM class is abstract or concrete
- Constraints on the OM attributes

A complete description of OM classes, OM attributes, syntaxes, and values that are defined for XDS and XOM APIs are described in “Part 4. XDS/XOM Supplementary Information” on page 179. Tables and textual descriptions, such as the one shown in Figure 29 for the concrete OM class `DS_C_ATTRIBUTE`, are provided for each OM class.


Class name
Description of the class including an indication if it is an abstract class
Indicates super-classes
Table showing values of syntax, length, number of values, and initial value
Description of attributes and listing of attribute values

DS_C_ATTRIBUTE

An instance of OM class **DS_C_ATTRIBUTE** is an attribute of an object and, thus a component of its directory entry.

An instance of this OM class has the OM attributes of its superclass, **OM_C_OBJECT**, in addition to the OM attributes listed in the following table.

OM_Atrributes of a DS_C_ATTRIBUTE

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_ATTRIBUTE_TYPE	String **(OM_S_OBJECT_ IDENTIFIER_STRING)**	-	1	-
DS_ATTRIBUTE_VALUES	any	-	0 or more	-

° **DS_ATTRIBUTE _TYPE**

The attribute type that indicates the class of information given by this attribute.

° **DS_ATTRIBUTE_VALUES**

The attribute values. The OM value syntax and the number of the values allowed for
this
OM attribute are determined by the value of the **DS_ATTRIBUTE_TYPE** OM attribute
in
accordance with the rules given in "Attribute and AVA." If the values of this OM
attribute
have the syntax string(*), the strings can be long and segmented. For this reason,
om_read() and **om_write()** need to be used to access all Strings(*) values.

Note: A directory attribute must always have at least one value, although it is acceptable for instances of this OM class not to have any values.

Figure 29. Complete Description of Concrete OM Class DS_C_ATTRIBUTE

The table shown in Figure 29 provides information under the following headings:

- OM Attribute

  This is the name of each of the OM attributes.
- Value Syntax

  This provides the syntaxes of each of the OM attribute's values.
- Value Length

  This describes any constraints on the number of bits, octets, or characters in each value that is a string.
- Value Number

  This describes any constraints on the number of values.
- Value Initially

  This is any value with which the OM attribute can be initialized.

An OM class can be constrained to contain only one member of a set of OM attributes. In turn, OM attributes can be restricted to having no more than a fixed number of values, either 0 (zero) or 1 as an optional value, or exactly one mandatory value.

An OM attribute’s value may be also constrained to a single syntax. That syntax can be further restricted to a subset of defined values.

An object passed as a parameter to an XOM and XDS function call needs to meet a minimum set of conditions, as follows:
- The type of each OM attribute must be specific to the object’s OM class or one of its superclasses.
- The number of values of each OM attribute must be within OM class limits.
- The syntax of each value must be among those the OM class permits.
- The number of bits, octets, or characters in each string value must be within OM class limits.

Packages

A *package* is a collection of OM classes that are grouped together, usually by function. The packages themselves are features that are negotiated with the directory service by using the XDS function **ds_version( )**. Consider which OM classes will be required for your application programs and determine the packages that contain these OM classes.

A package is uniquely identified by an ASN.1 object identifier. DCE XDS API supports the following four packages, where one is mandatory and three are optional:
- The directory service package (mandatory)
- The basic directory contents package (optional)
- The GDS package (optional)
- The message handling system (MHS) directory user package (optional)

The Directory Service Package

The directory service package is the default package and as such does not require negotiation. The optional packages have to be negotiated with the directory service by using the **ds_version( )** function.

The object identifiers for specific packages are defined in header files that are part of the XDS API and XOM API. An object identifier consists of a string of integers. The header files include `#define` preprocessor statements that assign names to the constants in order to make them more readable. For the application programmer, these assignments alleviate the burden of maintaining strings of integers. For example, the object identifiers for the directory service package are defined in **xds.h**. The **xds.h** header file contains OM class and OM attribute names, OM object constants, and defines prototypes for XDS API functions, as shown in the following code fragment from **xds.h**:

```
/* DS package object identifier */
/* {iso(1) identifier-organization(3) icd-ecma(12)
 *  member-company(2)
 *  dec(1011) xopen(28) dsp(0) } */

#define OMP_O_DS_SERVICE_PKG  "\x2B\x0C\x02\x87\x1C\x00"
```

A **ds_version( )** function call must be included within an application program to negotiate the optional features (packages) with the directory service. The first step is to build an array of object identifiers for the optional packages to be negotiated (the basic directory contents package and the GDS package), as shown in the following code fragment from the **acl.h** header file:

```
DS_feature features[] = {
 { OM_STRING(OMP_O_DS_BASIC_DIR_CONTENTS_PKG), OM_TRUE },
 { OM_STRING(OMP_O_DSX_GDS_PKG), OM_TRUE },
 { 0 }
};
```

The `OM_STRING` macro is provided for creating a data value of data type `OM_string` for octet strings and characters. XOM API macros are described in "XOM API Macros" on page 119.

The array of object identifiers is stored in `features` and passed as an input parameter to ds_version( ), as shown in the following code fragment from **acl.c**:

```
/* Negotiate the use of the BDC packages. */

if (ds_version(features) != DS_SUCCESS)
   printf("ds_version()error\n");
```

The Basic Directory Contents Package

The basic directory contents package contains the object identifier definition of directory classes and attribute types as defined by the X.500 standard. These definitions allow the creation of and maintenance of directory entries for a number of common objects so that the representation of all such objects is the same throughout the directory. Also included are the definitions of the OM classes and OM attributes required to support the directory attribute types. "Chapter 11. Basic Directory Contents Package" on page 219 describes the basic directory contents package in detail.

The object identifier associated with the basic directory contents package is shown in the following code fragment from the **xdsbdcp.h** header file:

```
/* BDC package object identifier */
/* { iso(1) identifier-organization(3) icd-ecma(12)
 *   member-company (2)
 *   dec(1011) xopen(28) bdcp(1) } */

#define OMP_DS_BASIC_DIR_CONTENTS_PKG"\x2B\x0C\x02\x87\x73\x1C\x01"
```

Note: The **xdsbdcp.h** header file must be included.

Package Closure

An OM class can be defined to have an attribute whose OM class is defined in some other package in order to avoid duplication of OM classes. This gives rise to the concept of a package closure. A package closure is the set of all OM classes that need to be supported so that all possible instances of all OM classes can be defined in the package.

Workspaces

Two application-specific APIs or two different implementations of the same service require work areas, called *workspaces*, to maintain private and public (service-generated) objects. The workspace is required because two implementations of the same service (or different services) can represent private objects differently. Each one has its own workspace. Using the functions provided by XOM API, such as **om_get(**) and **om_copy( )**, objects can be copied and moved from one workspace to another.

Recall that private objects are returned by a service to a workspace in private implementation-specific format. Using the OM function calls described in “OM Function Calls” on page 114, the data can be extracted from the private object for further program processing.

Before a request to the directory can be made by an application program, a workspace must be created by using the appropriate XDS function. An application creates a workspace by performing the XDS API call **ds_initialize( )**. Once the workspace is obtained, subsequent XDS API calls, such as **ds_read( )**, return a pointer to a private object in the workspace. When program processing is completed, the workspace is destroyed by using the **ds_shutdown( )** XDS API function. Implicit in **ds_shutdown( )** is a call to the XOM API function **om_delete( )** to delete each private object the workspace contains.

The programs located in **/opt/dcelocal/examples/xdsxom** demonstrate how to initialize and shut down a workspace. The XDS functions **ds_initialize( )** and **ds_shutdown( )** are described in detail in “Chapter 6. XDS Programming” on page 123.

The closures of one or more packages are associated with a workspace. A package can be associated with any number of workspaces. An application program must obtain a workspace that supports an OM class before it is able to create any instances of that OM class.

Storage Management

An object occupies storage. The storage occupied by a public object is allocated by the client and can therefore be directly accessed and released by the client. The storage occupied by a private object is not accessible by the client and must be managed indirectly by using XOM function calls. Release of service-generated public objects must also be managed indirectly by the client using the **om_delete( )** function.

Objects are accessed by an application program via object handles. Object handles are used as input parameters to interface functions by the client and returned as output parameters by the service. The object handle for a public object is simply a pointer to the data structure (an array of descriptors) containing the object OM attributes. The object handle for a private object is a pointer to a data structure that is in private implementation-specific format and, therefore, inaccessible directly by the client.

The client creates a client-generated public object by using normal programming language constructs; for example, static initialization. The client is responsible for managing any storage involved. The service creates service-generated public objects and allocates the necessary storage. As previously mentioned, the client must destroy service-generated public objects and release the storage by applying the XOM function **om_delete( )** to it, as shown in the following code fragment:

```
/*  We can now safely release all the private objects
 *  and the public objects we no longer need
 */
CHECK_OM_CALL(om_delete(session));
CHECK_OM_CALL(om_delete(result));
CHECK_OM_CALL(om_delete(entry));
CHECK_OM_CALL(om_delete(attributes));
CHECK_DS_CALL(ds_shutdown(workspace));
```

The service also creates private objects for which it allocates storage that must be managed by the application.

One of the input parameters to the **ds_read( )** function call is *name*. The *name* parameter is a public object created by the application from a series of nested data structures (RDNs and AVAs) to represent the distinguished name containing `Peter Piper`. When the application no longer needs the public object, it issues the XDS function call **ds_shutdown( )** to release the memory resources associated with the public object. The **ds_read( )** call returns the pointer to a private object, `result`, deposited in the workspace by the service.

The program goes on to use the XOM function **om_get( )** with the input parameter *result* as a pointer to extract attribute values from the returned private object. The **om_get( )** call returns the pointer *entry* as a service-generated public object to the program so that the attribute values specified in the call can be accessed by it. Once the value is extracted, the application can continue processing; for example, printing a message to a user with some extracted value like a phone number or postal address. The service-generated public object becomes the responsibility of the application program. The program goes on to release the resources allocated by the service by issuing a series of calls to **om_delete( )**, as shown in the following code fragment from `example.h`:

```
/*
 * extract the telephone number(s) of "name" from the result
 *
 * There are 4 stages:
 * (1) get the Entry-Information from the Read-Result.
 * (2) get the Attributes from the Entry-Information.
 * (3) get the list of phone numbers.
 * (4) scan the list and print each number.
 */

CHECK_OM_CALL(   om_get(result,
             OM_EXCLUDE_ALL_BUT_THESE_TYPES
           + OM_EXCLUDE_SUBOBJECTS,
             entry_list, OM_FALSE, 0, 0, &entry,
             &total_num));

CHECK_OM_CALL(   om_get(entry->value.object.object,
             OM_EXCLUDE_ALL_BUT_THESE_TYPES
           + OM_EXCLUDE_SUBOBJECTS,
             attributes_list, OM_FALSE, 0, 0, &attributes,
             &total_num));

CHECK_OM_CALL(   om_get(attributes->value.object.object,
             OM_EXCLUDE_ALL_BUT_THESE_TYPES
           + OM_EXCLUDE_SUBOBJECTS,
             telephone_list, OM_FALSE, 0, 0, &telephones,
             &total_num));
```

```
/*  We can now safely release all the private objects
 *  and the public objects we no longer need
 */
CHECK_OM_CALL(om_delete(session));
CHECK_OM_CALL(om_delete(result));
CHECK_OM_CALL(om_delete(entry));
CHECK_OM_CALL(om_delete(attributes));
```

If the client possesses a valid handle (or pointer) for an object, it has access to a private object. If the client does not possess an object handle or the handle is not a valid one, a private object is inaccessible to the client and an error is returned to the calling function. In the preceding code fragment, the handles for the objects stored in *entry*, *attributes*, and *telephones* are the pointers *&entry*, *&attributes*, and *&telephones*, respectively.

OM Syntaxes for Attribute Values

An OM attribute is made up of an integer uniquely defined within a package that indicates the OM attribute's type, an integer giving that value's syntax, and an information item called a *value*. The syntaxes defined by the XOM API standard are closely aligned with ASN.1 types and type constructors.

Some syntaxes are described in the standard in terms of syntax templates.

A syntax template defines a group of related syntaxes. The syntax templates that are defined are as follows:

- Enum(*)
- Object(*)
- String(*)

Enumerated Types

An OM attribute with syntax template Enum(*) is an enumerated type (OM_S_ENUMERATION) and has a set of values associated with that OM attribute. For example, one of the OM attributes of the OM class DS_C_ENTRY_INFO_SELECTION is DS_INFO_TYPE. DS_INFO_TYPE is listed in the OM attribute table for DS_C_ENTRY_INFO_SELECTION in "Chapter 10. XDS Class Definitions" on page 189 as having a value syntax of Enum(DS_Information_Type), as shown in Table 12. DS_INFO_TYPE takes one of the following values:

- DS_TYPES_ONLY
- DS_TYPES_AND_VALUES

Table 12. Description of an OM Attribute By Using Syntax Enum(*)

OM Attributes of DS_C_ENTRY_INFO_SELECTION				
OM Attribute	**Value Syntax**	**Value Length**	**Value Number**	**Value Initially**
DS_ALL_ATTRIBUTES	OM_S_BOOLEAN	—	1	OM_TRUE
DS_ATTRIBUTES_ SELECTED	String(OM_S_OBJECT_ IDENTIFIER_STRING)	—	0 or more	—
DS_INFO_TYPE	Enum(DS_Information_ Type)	—	1	DS_ TYPES_ AND_ VALUES

The C language representation of the syntax of the OM attribute type DS_INFO_TYPE is OM_S_ENUMERATION as defined in the **xom.h** header file. The value of the OM attribute is either DS_TYPES_ONLY or DS_TYPES_AND_VALUES, as shown in the following code fragment from example.h:

```
/*
 * Public Object ("Descriptor List") for
 * Entry-Information-Selection
 * parameter to ds_read().
 */
 OM_descriptor selection[] = {
 OM_OID_DESC(OM_CLASS, DS_C_ENTRY_INFO_SELECTION),
 { DS_ALL_ATTRIBUTES, OM_S_BOOLEAN, { OM_FALSE, NULL } },
 OM_OID_DESC(DS_ATTRIBUTES_SELECTED, DS_A_PHONE_NBR),
 { DS_INFO_TYPE,OM_S_ENUMERATION,
 { DS_TYPES_AND_VALUES,NULL } },
 OM_NULL_DESCRIPTOR
 };
```

Object Types

An OM attribute with syntax template Object(*) has OM_S_OBJECT as syntax and a subobject as a value. For example, one of the OM attributes of the OM class DS_C_DS_DN is DS_RDNS. DS_RDNS is listed in the OM attribute table for DS_C_DS_DN as having a value syntax of Object(DS_C_DS_RDN), as shown in Table 13.

Table 13. Description of an OM Attribute By Using Syntax Object(*)

OM Attributes of DS_C_DS_DN				
OM Attribute	**Value Syntax**	**Value Length**	**Value Number**	**Value Initially**
DS_RDNS	Object(DS_C_DS_RDN)	—	0 or more	—

The C language representation of the syntax of the OM attribute type DS_RDNS is OM_S_OBJECT, as shown in following code fragment from example.h:

```
OM_descriptor       name[] = {
 OM_OID_DESC(OM_CLASS, DS_C_DS_DN),
 { DS_RDNS, OM_S_OBJECT, { 0, rdn1 } },
 { DS_RDNS, OM_S_OBJECT, { 0, rdn2 } },
 { DS_RDNS, OM_S_OBJECT, { 0, rdn3 } },
 { DS_RDNS, OM_S_OBJECT, { 0, rdn4 } },
 OM_NULL_DESCRIPTOR
 };
```

Strings

An OM attribute with syntax template String(*) specifies the string syntax of its value. A string is categorized as either a *bit string*, an *octet string*, or a *character string*. The bits of a bit string, the octets of an octet string, or the octets of a character string constitute the *elements* of the string. (Refer to “Chapter 12. Information Syntaxes” on page 233 for a list of the syntaxes that form the string group.)

The value length of a string is the number of elements in the string. Any constraints on the value length of a string are specified in the appropriate OM class definitions.

The elements of the string are numbered. The position of the first element is 0 (zero). The positions of successive elements are successive positive integers.

For example, one of the attributes of the OM class DS_C_ENTRY_INFO_SELECTION is DS_ATTRIBUTES_SELECTED. DS_ATTRIBUTES_SELECTED is listed in the OM attribute table for DS_C_ENTRY_INFO_SELECTION as having a value syntax of String(OM_S_OBJECT_IDENTIFIER_STRING), as shown in Table 12 on page 108.

Other Syntaxes

The other syntaxes are defined as follows:

- OM_S_BOOLEAN

  A value of this syntax is a Boolean; that is, the value can be OM_TRUE or OM_FALSE.
- OM_S_INTEGER

  A value of this syntax is a positive or negative integer.
- OM_S_NULL

  The one value of this syntax is a valueless placeholder.

Service Interface Data Types

The local variables within an application program that contain the parameters and results of XDS and XOM API function calls are declared by using a standard set of data types. These data types are defined by typedef statements in the **xom.h** header files. Some of the more commonly used data types are described in the following subsections. A complete description of service interface data types is provided in "Chapter 13. XOM Service Interface" on page 239 and in the *IBM DCE Version 3.1 for AIX and Solaris: Application Development Reference* .

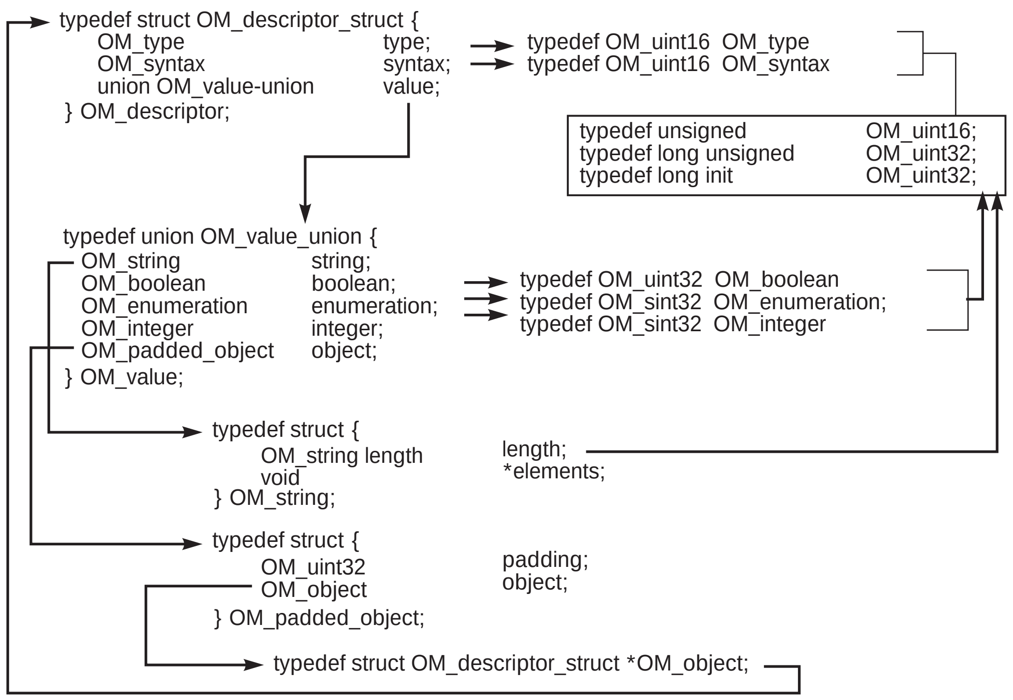
The OM_descriptor Data Type

The OM_descriptor data type is used to describe an OM attribute type, syntax, and value. A data value of this type is a descriptor, that embodies an OM attribute value. An array of descriptors can represent all the values of an object.

OM_descriptor is defined in the **xom.h** header file as follows:

```
/* Descriptor */

typedef struct OM_descriptor_struct {
        OM_type                 type;
        OM_syntax               syntax;
        union OM_value_union    value;
} OM_descriptor;
```

OM_descriptor is made up of a series of nested data structures, as shown in Figure 30 on page 111.



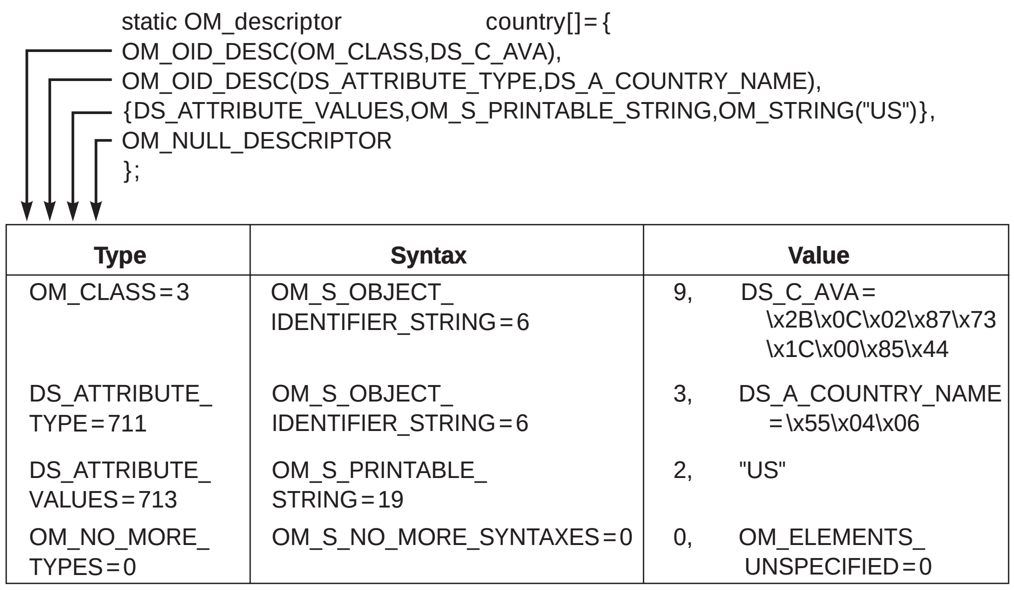
```
typedef struct OM_descriptor_struct {
      OM_type                         type;
      OM_syntax                       syntax;
      union OM_value-union            value;
 } OM_descriptor;

typedef OM_uint16  OM_type
typedef OM_uint16  OM_syntax

typedef unsigned              OM_uint16;
typedef long unsigned         OM_uint32;
typedef long init             OM_uint32;

typedef union OM_value_union {
   OM_string              string;
   OM_boolean             boolean;
   OM_enumeration         enumeration;
   OM_integer             integer;
   OM_padded_object       object;
 } OM_value;

typedef OM_uint32  OM_boolean
typedef OM_sint32  OM_enumeration;
typedef OM_sint32  OM_integer

typedef struct {
        OM_string length              length;
        void                          *elements;
} OM_string;

typedef struct {
        OM_uint32                     padding;
        OM_object                     object;
} OM_padded_object;

    typedef struct OM_descriptor_struct *OM_object;
```

Figure 30. Data Type OM_descriptor_struct

Figure 30 shows that `type` and `syntax` are integer constants for an OM attribute type and syntax, as shown in the following code fragment from `example.c`:

```
static OM_descriptor        country[] = {
 OM_OID_DESC(OM_CLASS, DS_C_AVA),
 OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_COUNTRY_NAME),
 { DS_ATTRIBUTE_VALUES,OM_S_PRINTABLE_STRING,OM_STRING("US") },
 OM_NULL_DESCRIPTOR
 };
```

The code fragment initializes four descriptors, as shown in Figure 31 on page 112. The type and syntax evaluate to integers for all four descriptors.



```
static OM_descriptor            country[]={
OM_OID_DESC(OM_CLASS,DS_C_AVA),
OM_OID_DESC(DS_ATTRIBUTE_TYPE,DS_A_COUNTRY_NAME),
{DS_ATTRIBUTE_VALUES,OM_S_PRINTABLE_STRING,OM_STRING("US")},
OM_NULL_DESCRIPTOR
};
```

Type	Syntax	Value
OM_CLASS=3	OM_S_OBJECT_ IDENTIFIER_STRING=6	9, DS_C_AVA= \x2B\x0C\x02\x87\x73 \x1C\x00\x85\x44
DS_ATTRIBUTE_ TYPE=711	OM_S_OBJECT_ IDENTIFIER_STRING=6	3, DS_A_COUNTRY_NAME =\x55\x04\x06
DS_ATTRIBUTE_ VALUES=713	OM_S_PRINTABLE_ STRING=19	2, "US"
OM_NO_MORE_ TYPES=0	OM_S_NO_MORE_SYNTAXES=0	0, OM_ELEMENTS_ UNSPECIFIED=0

Figure 31. Initializing Descriptors

The `value` component is a little more complex. Figure 30 on page 111 shows that `value` is a union of `OM_value_union`. `OM_value_union` has five members: `string`, `boolean`, `enumeration`, `integer`, and `object`. The members `boolean`, `enumeration`, and `integer` have integer values. The `string` member contains a string of type `OM_string`, which is a structure composed of a length and a pointer to a string of characters. The `object` member is a structure of type `OM_padded_object` that points to another object nested below it. Many OM attributes have other objects as values. These subobjects, in turn, may have other subobjects and so on.

For example, as shown in Figure 32, the OM class `DS_C_READ_RESULT` has one OM attribute: `DS_ENTRY`. The syntax of `DS_ENTRY` is `OM_S_OBJECT` with a value of `DS_C_ENTRY_INFO`, indicating that it points to the subobject `DS_C_ENTRY_INFO`. `DS_C_ENTRY_INFO` has the OM attribute `DS_OBJECT_NAME` with the syntax `OM_S_OBJECT`, indicating that it points to the subobject `DS_C_NAME`.

OM Class	Attribute	Syntax and Value
DS_C_READ_RESULT	DS_ENTRY	Objects(DS_C_ENTRY_INFO)
DS_C_ENTRY_INFO	DS_FROM_ENTRY DS_OBJECT_NAME	OM_S_BOOLEAN Object(DS_C_NAME)

Figure 32. An Object and a Subordinate Object

Data Types for XDS API Function Calls

The following code fragment from `example.h` shows how the data types are used to declare the variables that contain the output parameters from the XDS API function calls.

```
int main(void)
{
  DS_status         error;      /* return value from DS functions   */
  OM_return_code    return_code;/* return value from OM functions   */
  OM_workspace      workspace;  /* workspace for objects            */
  OM_private_object session;    /* session for directory operations */
  OM_private_object result;     /* result of read operation         */
  OM_sint           invoke_id;  /* Invoke-ID of the read operation
*/

CHECK_DS_CALL((OM_object) !(workspace=ds_initialize()));
  CHECK_DS_CALL(ds_version(bdcp_package, workspace));
  CHECK_DS_CALL(ds_bind(DS_DEFAULT_SESSION, workspace, &session));
```

The code fragment shows the following:

- The **ds_initialize( )** call returns a variable of type `OM_workspace` that contains a handle or pointer to a workspace.
- The **ds_bind( )** call returns a pointer to a variable of type `OM_private_object`. The private object contains the session information required by all subsequent XDS API calls, except **ds_shutdown( )**.
- The **ds_read( )** call returns a pointer to the result of a directory read request in a variable of type `OM_private_object`.
- The error handing macros `CHECK_DS_CALL` and `CHECK_OM_CALL`, defined in the `example.h` header file, use the data types `DS_status` and `OM_return_code`, respectively, as return values from XDS and XOM API function calls.

Data Types for XOM API Calls

The following code fragment from `example.h` shows how the data types are used to declare the variables that contain the input and output parameters for the XOM API function calls.

```
/*
 * variables to extract the telephone number(s)
 */
 OM_type            entry_list[]      = { DS_ENTRY, 0 };
 OM_type            attributes_list[] = { DS_ATTRIBUTES, 0 };
 OM_type            telephone_list[]  = { DS_ATTRIBUTE_VALUES, 0 };
 OM_public_object   entry;
 OM_public_object   attributes;
 OM_public_object   telephones;
 OM_descriptor     *telephone;  /* current phone number  */
 OM_value_position  total_num;  /* number of Attribute Descriptors
*/
```

The code fragment shows the following:

- The series of **om_get( )** calls requires a list of OM attribute types that identifies the types of OM attributes to be included in the operation. The variables `entry_list`, `attribute_list`, and `telephone_list` are declared as type `OM_type`.
- The series of **om_get( )** calls return pointers to variables of type `OM_public_object`. The **om_get( )** call generates public objects that are accessible to the application program.
- Where the variable `total_num` is type `OM_value_position` and is used to hold the number of OM descriptors returned by **om_get( )**.

“Chapter 12. Information Syntaxes” on page 233 contains detailed descriptions of all the data types defined by XOM API.

OM Function Calls

XOM API supports general-purpose OM functions defined by the X/Open standards body that allow an application program to manipulate objects in a workspace. "Summary of OM Function Calls" lists the OM function calls and gives a brief description of each. "Using the OM Function Calls" on page 115 illustrates the use of OM function calls by using the **om_get( )** call as an example.

Summary of OM Function Calls

The following list of XOM API function calls contains a brief description of each function. Refer to the appropriate reference page in the *IBM DCE Version 3.1 for AIX and Solaris: Application Development Reference*for a detailed description of the input and output parameters, return codes, and usage of each function.

- **om_copy( )**

  Creates an independent copy of an existing private object and all of its subobjects in a specified workspace.
- **om_copy_value( )**

  Replaces an existing OM attribute value or inserts a new value into a target private object with a copy of an existing OM attribute value found in a source private object.
- **om_create( )**

  Creates a private object that is an instance of the specified OM class.
- **om_delete( )**

  Deletes a private or service-generated public object.
- **om_get( )**

  Creates a new public object that is an exact, but independent, copy of an existing private object; certain exclusions and/or syntax conversion may be requested for the copy.
- **om_instance( )**

  Tests to determine if an object is an instance of a specified OM class (includes the case when the object is a subclass of that OM class).
- **om_put( )**

  Places or replaces copies of the attribute values of the source private or public object into the target private object.
- **om_read( )**

  Reads a segment of a string attribute from a private object.
- **om_remove( )**

  Removes and discards values of an attribute of a private object.
- **om_write( )**

  Writes a segment of a string attribute to a private object.
- **om_encode( )**

  Not supported by DCE XOM API.
- **om_decode( )**

  Not supported by DCE XOM API.

Using the OM Function Calls

Most application programs require the use of a series of **om_get( )** function calls to create service-generated public objects from which the program can extract requested information. For this reason, this section uses the operation of **om_get( )** as an example to describe how XOM API functions operate in general.

The following code fragment from `example.h` shows how a series of **om_get( )** function calls extract a list of telephone numbers from a workspace. The **ds_read( )** function call deposits the private object stored in `result` in the workspace and provides access to it by the pointer `&result`.

```
/*
 * extract the telephone number(s) of "name" from the result
 *
 * There are 4 stages:
 * (1) get the Entry-Information from the Read-Result.
 * (2) get the Attributes from the Entry-Information.
 * (3) get the list of phone numbers.
 * (4) scan the list and print each number.
 */

CHECK_OM_CALL(   om_get(result,
             OM_EXCLUDE_ALL_BUT_THESE_TYPES
           + OM_EXCLUDE_SUBOBJECTS,
             entry_list, OM_FALSE, 0, 0, &entry,
             &total_num));

CHECK_OM_CALL(   om_get(entry->value.object.object,
             OM_EXCLUDE_ALL_BUT_THESE_TYPES
           + OM_EXCLUDE_SUBOBJECTS,
             attributes_list, OM_FALSE, 0, 0, &attributes,
             &total_num));

CHECK_OM_CALL(   om_get(attributes->value.object.object,
             OM_EXCLUDE_ALL_BUT_THESE_TYPES
           + OM_EXCLUDE_SUBOBJECTS,
             telephone_list, OM_FALSE, 0, 0, &telephones,
             &total_num));

/*  We can now safely release all the private objects
 *  and the public objects we no longer need
 */
CHECK_OM_CALL(om_delete(session));
CHECK_OM_CALL(om_delete(result));
CHECK_OM_CALL(om_delete(entry));
CHECK_OM_CALL(om_delete(attributes));

for (telephone = telephones;
     telephone->type != DS_ATTRIBUTE_VALUES;
     telephone++)
    {
 if (telephone->type   != DS_ATTRIBUTE_VALUES
||  (telephone->syntax & OM_S_SYNTAX) !=
     OM_S_PRINTABLE_STRING)
      {
(void) fprintf(stderr, "malformed telephone number\n");
        exit(EXIT_FAILURE);
      }

(void) printf("Telephone number: %s\n",
        telephone->value.string.elements);
      }
```

```
CHECK_OM_CALL(om_delete(telephones));
CHECK_DS_CALL(ds_shutdown(workspace));
```

The **om_get( )** call makes a copy of all or a selected set of parts of a private object. The copy is a service-generated public object that is accessible to the application program. The application program extracts the list of telephone numbers from this copy.

Required Input Parameters

The **om_get( )** function requires the following input parameters:

- A private object
- A set of exclusions
- A set of OM attributes to be included in the copy
- A flag to indicate whether local string processing is required
- The position of the first value to be copied (the base value)
- The position within each OM attribute that is one beyond the last attribute to be included in the copy (indicating the scope of the copy)

The **om_get( )** call returns the following output parameters:

- The public object that is a copy of the private object
- The number of OM attribute descriptors returned in the public object

In the code fragment from `example.h`, the private object `result` is input to **om_get( )**.

The next parameter, the *exclusions* parameter, reduces the copy to a prescribed portion of the original. The exclusions apply to the OM attributes of the object, but not to those of subobjects. The possibilities for determining the combinations of types, values, subobjects, and descriptors to be excluded depend on the creativity of the programmer. For a detailed description of all the exclusion possibilities, refer to the *IBM DCE Version 3.1 for AIX and Solaris: Application Development Reference*. The values chosen for the **om_get( )** calls in `example.h` are simplified for clarity. These exclusion values are as follows:

- `OM_EXCLUDE_ALL_BUT_THESE_TYPES`
- `OM_EXCLUDE_SUBOBJECTS`

Each value indicates an exclusion, as defined by **om_get( )**, and is chosen from the set of exclusions; alternatively, the single value `OM_NO_EXCLUSIONS` may be chosen, which selects the entire object. Each value, except `OM_NO_EXCLUSIONS`, is represented by a distinct bit, the presence of the value being represented as 1, and its absence as 0 (zero). Multiple exclusions are requested by adding or ORing the values that indicate the individual exclusions.

`OM_EXCLUDE_ALL_THESE_TYPES` indicates that the OM attributes included are only the ones defined in the list of included types supplied in the next parameter, *entry_list*. `OM_EXCLUDE_SUBOBJECTS` indicates that, for each value whose syntax is `OM_S_OBJECT`, a descriptor containing an object handle for the original private subobject is returned, rather than a public copy of it. This handle makes that subobject accessible for use in subsequent function calls. Exclusion provides a means to examine an object one level at a time. The object the handle points to is used in the next **om_get( )** call to get the next level.

The *entry_list* parameter is declared in `example.h` as data type `OM_type` and initialized as a two-cell array with values `DS_ENTRY` and a NULL terminator. `DS_ENTRY` specifies the single OM attribute type included for that **om_get( )** call. This call only limits processing to the one directory entry; only one entry was defined previously in the program — the distinguished name of `Peter Piper`. The 0 (zero) marks the end of the OM attribute list.

The next parameter, `OM_FALSE`, indicates that mapping to a local string format is not required. The next two parameters set the initial and limiting value to 0 (zero), meaning that no specific values are to be excluded.

The final two parameters are output parameters: *entry*, a pointer to a service-generated public object deposited by **om_get( )** in the workspace, and *total_num*, an integer. Both *entry* and *total_num* are available for examination by the application program.

Extracting the Data from the Read Result

The *entry* parameter contains the result of processing by **om_get( )** of the `read` parameter generated by the **ds_read( )** operation. A successful call to **ds_read( )** returns an instance of OM class `DS_C_READ_RESULT` in the private object *result*. `DS_C_READ_RESULT` contains the information extracted from the directory entry of the target object. Figure 33 shows the relationship of some of the superclasses, subclasses, and the OM attribute of `DS_C_READ_RESULT`. Consider Figure 33 as a partial map of the contents of `result`.

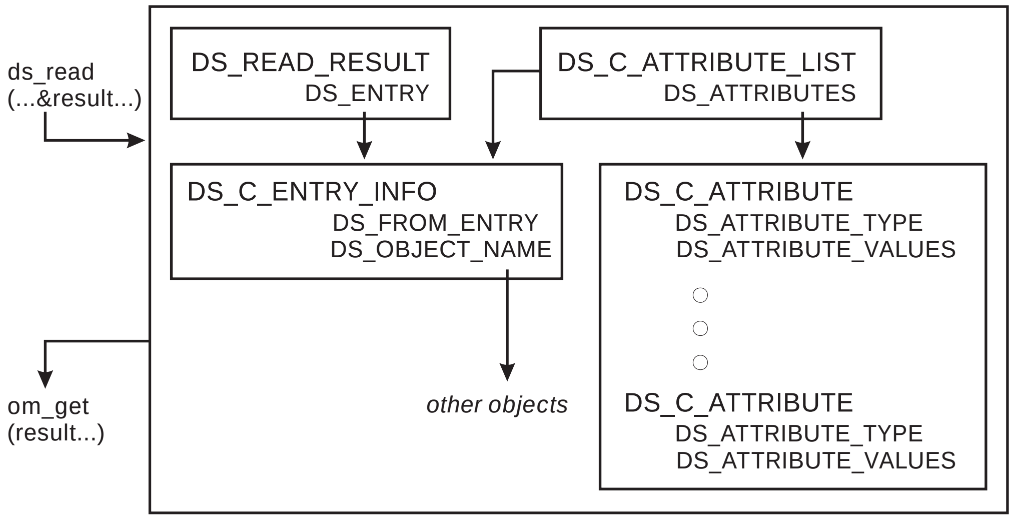


Figure 33. The Read Result

The **om_get( )** function call creates a public object to make the information contained in `result` available to the application program. The *entry* parameter is defined as data type `OM_public_object`. As such, it is composed of several nested layers of subobjects that contain entry information, OM attributes, and OM attribute values, as shown in Figure 34 on page 118. The series of **om_get( )** calls removes these layers of objects to extract a list of telephone numbers.

Figure 34 on page 118 also shows that the process of exposing the subobjects continues while the syntax of the subobjects is `OM_S_OBJECT`. In effect, `example.h` is reversing the process of building up a series of public objects as input to

ds_read( ); namely, the distinguished name of `Peter Piper` and the descriptor list for *entry_information_selection*.

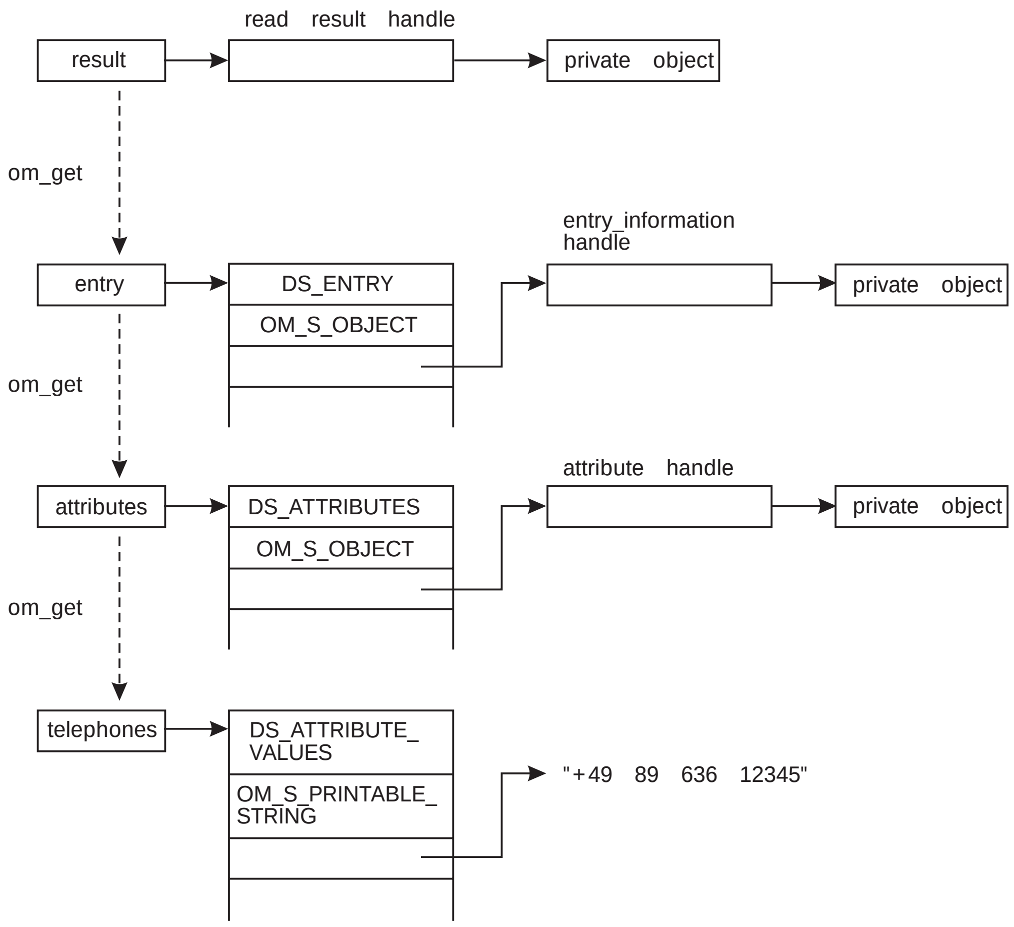


Figure 34. Extracting Information Using om_get( )

The following code fragment from `example.c` shows how the syntax of the variable `telephones` is tested for valid syntax; in this case, `OM_S_PRINTABLE_STRING`:

```
for (telephone = telephones;
     telephone->type != DS_ATTRIBUTE_VALUES;
     telephone++)
    {
 if (telephone->type != DS_ATTRIBUTE_VALUES ||
     (telephone->syntax & OM_S_SYNTAX) !=
     OM_S_PRINTABLE_STRING)
      {
(void) fprintf(stderr, "malformed telephone number\n");
        exit(EXIT_FAILURE);
      }
(void) printf("Telephone number: %s\n",
        telephone->value.string.elements);
      }
```

The preceding example determines whether `telephones` is in a format that can be used by the application program as string data that can be printed out, and that the syntax is correct for a list of telephone numbers. Note that the program uses the constant `OM_S_SYNTAX` to mask off the top 6 bits. These bits are special bits that are used by XOM API. (Refer to “Chapter 13. XOM Service Interface” on page 239 for more information on these special bits.)

Return Codes

XOM API function calls return a value of type `OM_return_code`, which indicates whether the function succeeded. If the function is successful, the value of `OM_return_code` is set to `OM_SUCCESS`. If the function fails, it returns one of the values listed in “Chapter 13. XOM Service Interface” on page 239. The constants for `OM_return_code` are defined in the **xom.h** header file.

XOM API Header Files

The XOM API includes the header file **xom.h**. This header file is composed of declarations defining the C workspace interface. It supplies type definitions, symbolic constant definitions, and macro definitions.

XOM Type Definitions and Symbolic Constant Definitions

The **xom.h** header file includes `typedef` statements that define the data types of all OM objects used in the interface. It also provides definitions of symbolic constants used by the interface.

Refer to the **xom.h(4xom)** reference page for more information.

XOM API Macros

XOM API provides several macros that are useful in defining public objects in your application programs. These macros are defined in the **xom.h** header file.

- `OM_IMPORT`

  Makes object identifier symbolic constants available within a C source module.
- `OM_EXPORT`

  Allocates memory and initializes object identifier symbolic constants within a C source module.
- `OM_OID_DESC`

  Initializes the type, syntax, and value of an OM attribute that holds an object identifier.
- `OM_NULL_DESCRIPTOR`

  Marks the end of a client-generated public object.
- `OMP_LENGTH`

  Calculates the length of an object identifier.
- `OM_STRING`

  Creates a data value of a string data type.

The OM_EXPORT and OM_IMPORT Macros

Most application programs find it convenient to export all the names they use from the same C source module. `OM_EXPORT` allocates memory for the constants that represent an object OM class or an object identifier, as shown in the following code fragment from `example.c`:

In this code fragment, object identifier constants that represent OM classes defined in the **xds.h** and **xdscds.h** header files are exported to the main program module. The object identifier constants are defined in **xds.h**, with the OMP_O prefix followed by the variable name for the object identifier. The constant itself provides the hexadecimal value of the object identifier string.

The OM_EXPORT macro takes the OM class name as input and creates two new data structures: a character string and a structure of type OM_string. The structure of type OM_string contains a length and a pointer to a string that may be used later in an application program by the OM_OID_DESC macro to initialize the value of an object identifier.

OM_IMPORT marks the identifiers as external for the compiler. It is used if OM_EXPORT is called in a different file from where its values are referenced. OM_IMPORT is not used in example.c because OM_EXPORT is called in the file where the object identifiers are referenced.

The OM_OID_DESC and OMP_LENGTH Macros

The OM_OID_DESC macro initializes the type, syntax, and value of an OM attribute that holds an object identifier; in other words, it initializes OM_descriptor. It takes as input an OM attribute type and the name of an object identifier. The object identifier should have already been exported to the program module, as shown in the previous section.

The output of the macro is an OM_descriptor composed of a type, syntax, and value. The type is the name of the OM class. The syntax is OM_S_OBJECT_IDENTIFIER. The value is a two-member structure with the length of the object identifier and a pointer to the actual object identifier string. It is defined as a pointer to void so that it can be used as a generic pointer; it can point to any data type.

OM_OID_DESC calls OMP_LENGTH to calculate the length of the object identifier string.

The following code fragment from **xom.h** shows the OM_OID_DESC and OMP_LENGTH macros:

```
/* Private macro to calculate length
 * of an object identifier
 */
#define OMP_LENGTH(oid_string) (sizeof(OMP_O_##oid_string)-1)

/* Macro to initialize the syntax and value
 * of an object identifier
 */
#define OM_OID_DESC(type, oid_name)
        { (type), OM_S_OBJECT_IDENTIFIER_STRING,
         { OMP_LENGTH(oid_name) , OMP_D_##oid_name }
}
```

The OM_NULL_DESCRIPTOR Macro

The OM_NULL_DESCRIPTOR macro marks the end of a client-generated public object by setting the type, syntax, and value to OM_NO_MORE_TYPES, OM_S_NO_MORE_SYNTAXES, and a value of zero length and a NULL string, respectively.

The OM_STRING Macro

The `OM_STRING` macro creates a string data value. Data strings are of type `OM_string`, as shown in this code fragment from the **xom.h** header file:

```
/* String */

typedef struct {
        OM_string_length     length;
        void                 *elements;
} OM_string;

#define OM_STRING(string)       \
         { (OM_string_length)(sizeof(string)-1), string
}
```

A string is specified in terms of its length or whether or not it terminates with a NULL. `OM_string_length` is the number of octets by which the string is represented, or it is the `OM_LENGTH_UNSPECIFIED` value if the string terminates with a NULL.

The bits of a bit string are represented as a sequence of octets. The first octet stores the number of unused bits in the last octet. The bits in the bit string, beginning with the first bit and proceeding to the trailing bit, are placed in bits 7 to 0 of the second octet. These are followed by bits 7 to 0 of the third octet, then by bits 7 to 0 of each octet in turn, followed by as many bits as are required of the final octet commencing with bit 7.

Chapter 6. XDS Programming

The XDS API defines an application programming interface to directory services in the X/Open Common Applications Environment as defined in *X/Open Portability Guide*. This interface is based on the 1988 CCITT X.500 Series of Recommendations and the ISO 9594 Standard. This joint standard is referred to from this point on simply as X.500.

This chapter describes the purpose and function of XDS API functions in a general way. Refer to the reference pages in the *IBM DCE Version 3.1 for AIX and Solaris: Application Development Reference* for complete and detailed information on specific function calls.

The sections that follow describe the following types of XDS functions:

- XDS interface management functions

  These functions interact with the XDS interface
- Directory connection management functions

  These functions initiate, manage, and terminate connections with the directory
- Directory operation functions

  These functions perform operations on a directory

Note: The DCE XDS API does not support asynchronous operations from within the same thread. If an application requires asynchronous XDS operations, then it should use multiple threads to achieve this functionality.

The **ds_abandon( )** function is not supported in this release. A **ds_abandon( )** call returns a `DS_C_ABANDON_FAILED` (`DS_E_TOO_LATE`) error. Refer to “Chapter 9. XDS Interface Description” on page 181 for information on abandoning directory operations.

The following names refer to the complete XDS example programs, located in the sample files located in the **/opt/dcelocal/examples/xdsxom** directory.

- **acl.c** (**acl.h**)
- `example.c` (`example.h`)
- **teldir.c**

XDS Interface Management Functions

XDS API defines a set of functions that only interact with the XDS interface and have no counterpart in the directory standard definition:

- **ds_initialize( )**
- **ds_version( )**
- **ds_shutdown( )**

These interface functions perform operations that involve the initialization, management, and termination of sessions with the XDS interface service.

The ds_initialize( ) Function Call

Every application program must first call **ds_initialize( )** to establish a workspace where objects returned by the directory service are deposited. The **ds_initialize( )** function must be called before any other directory interface functions are called.

The **ds_initialize( )** call returns a handle (or pointer) to a workspace. The application program performs operations on OM objects in this workspace. OM objects created in this workspace can be used as input parameters to the other directory interface functions. In addition, objects returned by the directory service are deposited in the workspace.

Within the following code fragment from `example.c`, a workspace is initialized. (The declaration of the variable *workspace* and the call to **ds_initialize( )** are found in different sections of the program.)

```
int main(void)
{
  DS_status         error;      /* return value from DS functions   */
  OM_return_code    return_code;/* return value from OM functions   */
  OM_workspace      workspace;  /* workspace for objects            */
  OM_private_object session;    /* session for directory operations */
  OM_private_object result;     /* result of read operation         */
  OM_sint           invoke_id;  /* Invoke-ID of the read operation  */
  OM_value_position total_num;  /* Number of Attribute Descriptors  */

/*
 * Perform the Directory operations:
 * (1) Initialize the directory service and get an OM workspace.
 * (2) bind a default directory session.
 * (3) read the telephone number of "name".
 * (4) terminate the directory session.
 */

  CHECK_DS_CALL((OM_object) !(workspace=ds_initialize()));
```

`OM_workspace` is a type definition in the **xom.h** header file defined as a pointer to `void`. A void pointer is a generic pointer that may point to any data type. The variable *workspace* is declared as data type `OM_workspace`. The return value is assigned to the variable *workspace*, and the `CHECK_DS_CALL` macro determines if the call is successful. `CHECK_DS_CALL` is an error-handling macro that is defined in `example.h`.

The **ds_initialize( )** call returns a handle to a workspace in which OM objects can be created and manipulated. Only objects created in this workspace can be used as parameters to other directory interface functions. The **ds_initialize( )** call returns NULL if it fails.

The ds_version( ) Function Call

The **ds_version( )** call negotiates features of the directory interface. These features are collected into packages that define the scope of the service. Packages define such things as object identifiers for directory and OM classes and OM attributes, enumerated types, structures, and OM object constants.

XDS API defines the following packages in separate header files as part of the XDS API software product:

- Directory service package

The directory service package contains the OM classes and OM attributes used to interact with the directory service. This package is contained in the **xds.h** header file.

- Basic directory contents package

  The basic directory contents package contains OM classes and OM attributes that represent values of selected attributes and selected objects defined in the X.500 standard. This package is contained in the **xdsbdcp.h** header file.

Note: 2.5.4.6 and 2.5.6.2 are object identifiers defined by the standards, not the BER encoding found in the header file which are `\x55\x04\x06` and `\x55\x06\x02`.

The ds_shutdown( ) Function Call

The **ds_shutdown( )** call deletes the workspace established by **ds_initialize( )** and enables the directory service to release resources. No other directory functions that reference that workspace may be called after this function.

The following code fragment from **example.h** demonstrates how the application closes the directory workspace by performing a **ds_shutdown( )** call.

```
CHECK_DS_CALL(ds_shutdown(workspace));
```

Directory Connection Management Functions

The following subsections describe the XDS functions that initiate, manage, and terminate connections with the directory service.

- “A Directory Session”
- “The ds_bind( ) Function Call”
- “The ds_unbind( ) Function Call” on page 126
- “Automatic Connection Management” on page 126

A Directory Session

A directory session identifies the DSA to which a directory operation is sent. It also defines the characteristics of a session, such as the distinguished name of the requestor.When using XDS/XOM over CDS, it is used to bind to the CDS namespace.

An application program can request a session with specific OM attributes tailored for the program’s requirements. The application passes an instance of OM class `DC_C_SESSION` with the appropriate OM attributes, or it uses the default parameters by passing the constant `DS_DEFAULT_SESSION` as a parameter to the **ds_bind( )** function call. **DS_DEFAULT_SESSION** is sufficient when using XDS/XOM over CDS.

The ds_bind( ) Function Call

The **ds_bind** call establishes a session with the CDS namespace. The **ds_bind( )** call corresponds to the `DirectoryBind` function in the Abstract Service defined in the X.500 standard.

When a **ds_bind( )** call completes successfully, the directory returns a pointer to an OM private object of OM class `DC_C_SESSION`. This parameter is then passed as the first parameter to most interface function calls until a **ds_unbind( )** is called to terminate the directory session.

XDS API supports multiple concurrent sessions so an application can interact with the directory service by using several identities, and interact directly and concurrently with different parts of the CDS namespace.

The following code fragment from `example.c` shows how an application binds to the CDS namespace by using the default session:

```
CHECK_DS_CALL(ds_bind(DS_DEFAULT_SESSION, workspace,&session));
```

The ds_unbind( ) Function Call

The **ds_unbind( )** call terminates a directory session and makes the *session* parameter unavailable for use with other interface functions. However, the unbound session can be modified by OM functions and used again as a parameter to **ds_bind( )**. When the *session* parameter is no longer needed, it should be deleted by using OM functions such as **om_delete( )**.

The following code fragment from `example.c` shows how the application closes the connection to the CDS namespace by using **ds_unbind( )**:

```
/* Close the connection to the GDS server.  */

if (ds_unbind(bound_session) != DS_SUCCESS)
    printf("ds_unbind() error\n");
```

The **ds_unbind( )** call corresponds to the `DirectoryUnbind` function in the Abstract Service defined in the X.500 standard.

Automatic Connection Management

The XDS implementation does not support automatic connection management. A CDS connection is established when **ds_bind( )** is called and released when **ds_unbind( )** is called.

XDS Interface Class Definitions

The XDS interface class definitions are described in detail in “Chapter 10. XDS Class Definitions” on page 189. The OM attribute types, syntax, and values and inheritance properties are described for each OM class.

A good way to begin to understand how the OM class hierarchy is structured and the relationship between OM classes and OM attributes to the service provided by the directory service package is to look up one of the OM classes listed in “Chapter 10. XDS Class Definitions” on page 189.

Example: The DS_C_ATTRIBUTE_LIST Class

For example, **DS_C_ATTRIBUTE_LIST** inherits the OM attributes from its superclass **OM_C_OBJECT**, as do all OM classes. **OM_C_OBJECT**, as defined in “Chapter 5. XOM Programming” on page 83, has one OM attribute, **OM_CLASS**, with the value of an object identifier string that identifies the numeric representation of the object's OM class. **DS_C_ATTRIBUTE_LIST** also has one OM attribute.

The purpose of **DS_C_ATTRIBUTE_LIST** is to define a list of attributes for an object. It has the **DS_ATTRIBUTES** OM attribute. The **DS_ATTRIBUTE** OM attribute has a value of a pointer to the **DS_C_ATTRIBUTE** OM object.

Example: The DS_C_FILTER Class

For example, `DS_C_FILTER` inherits the OM attributes from its superclass `OM_C_OBJECT`, as do all OM classes. `OM_C_OBJECT`, as defined in "Chapter 14. Object Management Package" on page 255, has one OM attribute, `OM_CLASS`, which has the value of an object identifier string that identifies the numeric representation of the object's OM class. `DS_C_FILTER`, on the other hand, has several OM attributes.

The purpose of `DS_C_FILTER` is to select or reject an object on the basis of information in its directory entry. It has the following OM attributes:

- `DS_FILTER_ITEMS`
- `DS_FILTERS`
- `DS_FILTER_TYPE`

Two of these OM attributes, `DS_FILTER_ITEMS` and `DS_FILTERS`, have values that are OM object classes themselves. The value of the OM attribute `DS_FILTER_ITEMS` is `DS_C_FILTER_ITEM`, which is an OM class. `DS_C_FILTER_ITEM` is a component of a filter and defines the nature of the filter. The value of the OM attribute `DS_FILTERS` is `DS_C_FILTER`, an OM class. Thus, `DS_FILTERS` defines a collection of filters. The OM attribute `DS_FILTER_TYPE` has a value that is an enumerated type, which takes one of the values `DS_AND`, `DS_OR`, or `DS_NOT`.

The DS_C_CONTEXT Parameter

The OM class `DS_C_CONTEXT` is the second parameter to every directory service request. `DS_C_CONTEXT` defines the characteristics of the directory service interaction that are specific to a particular directory service operation. These characteristics are divided into three categories of OM attributes: common parameters, service controls, and local controls.

Common parameters affect the processing of each directory service operation.

Service controls indicate how the directory service should handle requests. Included in this category are decisions about whether or not chaining is permitted, the priority of requests, the scope of referral (to DSAs within a country or within a DMD), and the maximum number of objects about which a function should return information.

Local controls include asynchronous support and automatic continuation; XDS does not currently support asynchronous operations from within the same thread. Applications requiring asynchronous use of the XDS/XOM API should use threads as defined in "Chapter 7. Using Threads With The XDS/XOM API" on page 145.

Note: Service Controls and Local Controls are not supported for XDS/XOM over CDS.

Directory Class Definitions

The X.500 standards define a number of attribute types and classes. These definitions allow the creation and maintenance of directory entries for a number of common objects so that the representation of all such objects is the same throughout the directory. The basic directory contents package contains OM classes and OM attributes that model the X.500 attribute types and classes.

The X.500 object classes and attributes are defined in the following documents published by CCITT. These are the objects and the associated attributes that will be the targets of directory service operations in your application programs:

- *The Directory: Selected Attributes Types (Recommendation X.520)*
- *The Directory: Selected Object Classes (Recommendation X.521)*

Table 14 describes the OM classes, OM attributes, and their object identifiers that model the X.500 objects and attributes. (See “Chapter 11. Basic Directory Contents Package” on page 219 for more tables with the same type of information.)

Table 14. Representation of Values for Selected Attribute Types

Attribute Type	**OM Value Syntax**	**Value Length**	**Multi-valued**	**Matching Rules**
DS_A_ALIASED_ OBJECT_NAME	Object(DS_C_NAME)	—	no	E
DS_A_BUSINESS_ CATEGORY	String(OM_S_ TELETEX_STRING)	1–128	yes	E, S
DS_A_COMMON_ NAME	String(OM_S_ TELETEX_STRING)	1–64	yes	E, S
DS_A_COUNTRY_ NAME	String(OM_S_ PRINTABLE_ STRING)[1]	2	no	E
DS_A_DESCRIPTION	String(OM_S_ TELETEX_STRING)	1–1024	yes	E, S

1 As permitted by ISO 3166.

The tables in “Chapter 11. Basic Directory Contents Package” on page 219 contain similar categories of information as do similar tables for the attributes defined in the directory service package. These information categories include the following:

- OM Value Syntax
- Value Length
- Multivalued
- Matching Rules

The OM Value Syntax column describes the structure of the values of an OM attribute. The Value Length column gives the range of lengths permitted for the string types. The Multivalued column indicates whether the attribute can have multiple values.

The CCITT standards define matching rules that are used for determining whether two values are equal, for ordering two values, or for identifying one value as a substring of another in directory service operations. These are indicated in the Matching Rules column.

Note: The above elements are used when the cell name is an X.500 style name. When using XDS/XOM over CDS, none of the checking for value length and related information is performed. Only attribute types like `DS_A_COUNTRY_NAME` used in the cell name should be required from the Basic Directory Contents Package.

Directory Operation Functions

The X.500 standard defines the operations provided by the directory in a document called the *Abstract Service Definition*. DCE implements this standard with XDS API functions calls. The XDS API functions allow an application program to interact with the directory service. The standard divides these interactions into three general categories: read, search, and modify.

The XDS API functions correspond to the Abstract Service functions defined in the X.500 standard, as shown in Table 14 on page 128.

Table 15. Mapping of XDS API Functions to the Abstract Services

`XDS Function Call`	`Equivalent Abstract Service`
ds_read( )	`Read`
ds_compare( )	`Compare`
ds_list( )	`List`
ds_search( )	`Search`
ds_add_entry( )	`AddEntry`
ds_remove_entry( )	`RemoveEntry`
ds_modify_entry( )	`ModifyEntry`
ds_modify_rdn( )	`ModifyRDN`

Note: **ds_search( )** and **ds_modify_rdn( )** are not supported for XDS/XOM over CDS.

Directory Read Operations

Read functions retrieve information from specific named entries in the directory where names are mapped to attributes. This is analogous to looking up some information about a name in the ″White Pages″ phone directory.

XDS API implements the following read functions:

- **ds_read( )**

  The requestor supplies a distinguished name and one or more attribute types. The value(s) of requested attributes or just the attribute type(s) is returned by the DSA.
- **ds_compare( )**

  The requestor gives a distinguished name and an attribute value assertion (AVA). If the AVA is TRUE for the named entry, a value of TRUE is returned by the DSA.

For example, a typical read operation could request the telephone number of a particular employee. A read request would submit the distinguished name of the employee with an indication to return its telephone number: `/C=us/O=sni/OU=sales/CN=John Smith`.

Reading an Entry from the Directory

The following sections describe a typical read operation by using the **ds_read( )** function call. They include a description of tasks directly related to the read operation. They do not include service-related tasks such as initializing the interface, allocating an OM workspace, and binding to the directory. These tasks are described in “XDS Interface Management Functions” on page 123. The following sections also do not describe the process of extracting information from the workspace by using XOM functions. Refer to “Chapter 5. XOM Programming” on page 83 for a description of how to use XOM functions to access the workspace.

A typical read operation involves the following steps:

1. Define the necessary object identifier constants for the OM classes and OM attributes that will define public objects for input to **ds_read( )**, by using the `OM_EXPORT` macro.
2. Declare the variables that will contain the output from the XDS functions to be used in the application.
3. Build public objects (descriptor lists) for the *name* parameter to **ds_read( )**.
4. Create a descriptor list for the *selection* parameter to **ds_read( )** that selects the type and scope of information in your request.
5. Perform the read operation.

These steps are demonstrated in the following code fragments from `example.c` (see the **/opt/dcelocal/examples/xdsxom** for a complete program listing). The program reads the telephone numbers of a given target entry.

Step 1: Export Object Identifiers for Required Directory Classes and Attributes

Most application programs find it convenient to export all the names they use from the same C source module. In the following code fragment from `example.c`, the `OM_EXPORT` macro allocates memory for the constants that represent the OM object classes and directory attributes required for the read operation:

```
/*  Define necessary Object Identifier constants */
OM_EXPORT(DS_A_COUNTRY_NAME)
OM_EXPORT(DS_A_ORG_NAME)
OM_EXPORT(DS_A_ORG_UNIT_NAME)
OM_EXPORT(DSX_TYPELESS_RDN)
OM_EXPORT(DS_C_AVA)
OM_EXPORT(DS_C_DS_DN)
OM_EXPORT(DS_C_DS_RDN)
OM_EXPORT(DS_C_ENTRY_INFO_SELECTION)
OM_EXPORT(DS_C_ATTRIBUTE)
OM_EXPORT(DS_C_ATTRIBUTE_LIST)
OM_EXPORT(DS_C_TELEPHONE_NUMBER)
```

The `OM_EXPORT` macro performs the following steps:

1. It defines a character array called `OMP_D_` concatenated with the *class_name* input parameter.

2. It initializes this array to the value of a character string called `OMP_O_` concatenated with the *class_name* input parameter. This value has already been defined in a header file.
3. It defines an `OM_string` data structure as the *class_name* input parameter.
4. It initializes the `OM_string` data structure's first component to the length of the array initialized in Step 2, and initializes the second component to a pointer to the value of the array initialized in Step 2.

Step 2: Declare Local Variables

The local variables *session*, *result*, and *invoke_id* are defined in the following code fragment from `example.c`:

```
int main(void)
{
  DS_status         error;      /* return value from DS functions  */
  OM_return_code    return_code;/* return value from OM functions  */
  OM_workspace      workspace;  /* workspace for objects           */
  OM_private_object session;    /* session for directory operations*/
  OM_private_object result;     /* result of read operation        */
  OM_sint           invoke_id;  /* Invoke-ID of the read operation */
  OM_value_position total_num;  /* Number of Attribute Descriptors */
```

These data types are defined in a `typedef` statement in the **xom.h** header file. The *session* and *result* variables are defined as data type `OM_private_object` because they are returned by **ds_bind( )** and **ds_read( )**, respectively, to the workspace as private objects. Since asynchronous operations (within the same thread) are not supported, the *invoke_id* functionality is redundant. The *invoke_id* parameter must be supplied to the XDS functions as described in the *IBM DCE Version 3.1 for AIX and Solaris: Application Development Reference*, but its return value should be ignored.

Values in *error* and *return_code* are returned by XOM and XDS functions to indicate whether a call was successful. The *workspace* variable is defined as `OM_workspace` and is used when establishing an OM workspace. The *total_num* variable is defined as `OM_value_position` to indicate the number of attribute descriptors returned in the public object by **om_get( )**, based on the inclusion and exclusion parameters specified.

Step 3: Build Public Objects

A **ds_read( )** function call can take a public object as an input parameter. A public object is generated by an application program and contains the information required to access a target directory object. This information includes the AVAs and RDNs that make up a distinguished name of an entry in the directory.

A public object is created by using OM classes and OM attributes. These OM classes and OM attributes model the target object entry in the directory and provide other information required by the directory service to access the directory. In this case, the target object entry in the directory is the entry for `Peter Piper`.

"Chapter 5. XOM Programming" on page 83 describes how to create the required public objects for the **ds_read( )** function call by using macros and data structures defined in the XDS and XOM API header files.

The purpose of building the public objects for AVAs and RDNs is to provide the public objects that represent a distinguished name. The distinguished name public object is stored in the array of descriptors called *name* and provided as an input parameter to the **ds_read( )** function call.

Step 4: Create an Entry-Information-Selection Parameter

The distinguished name for `Peter Piper` is an entry in the directory that the application is designed to access. The *selection* parameter of the **ds_read( )** function call tailors its results to obtain just part of the required entry. Information on all attributes, no attributes, or a specific group of attributes can be chosen. Attribute types are always returned, but the attribute values need not be.

The value of the parameter is a public object (descriptor list) that is an instance of OM class `DS_C_ENTRY_INFO_SELECTION`, as shown in the following code fragment from `example.c`:

```
/*
 * Public Object ("Descriptor List") for
 * Entry-Information-Selection
 * parameter to ds_read().
 */
 OM_descriptor selection[] = {
 OM_OID_DESC(OM_CLASS, DS_C_ENTRY_INFO_SELECTION),
 { DS_ALL_ATTRIBUTES, OM_S_BOOLEAN, { OM_FALSE, NULL } },
 OM_OID_DESC(DS_ATTRIBUTES_SELECTED, DS_A_PHONE_NBR),
 { DS_INFO_TYPE,OM_S_ENUMERATION,
 { DS_TYPES_AND_VALUES,NULL } },
 OM_NULL_DESCRIPTOR
 };
```

`DS_C_ENTRY_INFO_SELECTION` is a subclass of `OM_C_OBJECT`. (This information is supplied in the description of this class in “Chapter 10. XDS Class Definitions” on page 189.) As such, `DS_C_ENTRY_INFO_SELECTION` inherits the OM attributes of `OM_C_OBJECT`. The only OM attribute of `OM_C_OBJECT` is `OM_CLASS`. `OM_CLASS` identifies an object’s class, which in this case is `DS_C_ENTRY_INFO_SELECTION`. `DS_C_ENTRY_INFO_SELECTION` identifies information to be extracted from a directory entry and has the following OM attributes:

- `OM_C_CLASS` (inherited from `OM_C_OBJECT`)
- `DS_ALL_ATTRIBUTES`
- `DS_ATTRIBUTES_SELECTED`
- `DS_INFO_TYPE`

As part of a **ds_read( )** or **ds_search( )** function call, `DS_ALL_ATTRIBUTES` specifies to the directory service those attributes of a directory entry that are relevant to the application program. It can take the values `OM_TRUE` or `OM_FALSE`. These values are defined to be of syntax `OM_S_BOOLEAN`. The value `OM_TRUE` indicates that information is requested on all attributes in the directory entry. The value `OM_FALSE`, used in the preceding sample code fragment, indicates that information is only requested on those attributes that are listed in the OM attribute `DS_ATTRIBUTES_SELECTED`.

`DS_ATTRIBUTES_SELECTED` lists the types of attributes in the entry from which information is to be extracted. The syntax of the value is specified as `OM_S_OBJECT_IDENTIFIER_STRING`.

`OM_S_OBJECT_IDENTIFIER_STRING` contains an octet string of BER-encoded integers, which are decimal representations of object identifiers of the types of attributes in the attribute list. In the preceding code fragment, the string value is the attribute name DS_A_PHONE_NBR because the purpose of the read call is to read a list of telephone numbers from the directory.

DS_INFO_TYPE identifies what information is to be extracted from each attribute identified. The syntax of the value is specified as Enum(DS_Information_Type). DS_INFO_TYPE is an enumerated type that has two possible values: DS_TYPES_ONLY and DS_TYPES_AND_VALUES. DS_TYPES_ONLY indicates that only the attribute types of the selected attributes in the entry are returned by the directory service operation. DS_TYPES_AND_VALUES indicates that both the attribute types and the attribute values of the selected attributes in the entry are returned. The code fragment from example.c shown previously defines the value of DS_INFO_TYPE as DS_TYPES_AND_VALUES because the program wants to get the actual telephone numbers.

Step 5: Perform the Read Operation

The following code fragment from example.c shows the **ds_read( )** function call and the XDS calls that precede it:

```
/*
 * Perform the Directory operations:
 * (1) Initialize the directory service
 *     and get an OM workspace.
 * (2) bind a default directory session.
 * (3) read the telephone number of "name".
 * (4) terminate the directory session.
 */

CHECK_DS_CALL((OM_object) !(workspace = ds_initialize()));

CHECK_DS_CALL(ds_version(bdcp_package, workspace));

CHECK_DS_CALL(ds_bind(DS_DEFAULT_SESSION, workspace,
                &session));

CHECK_DS_CALL(ds_read (session, DS_DEFAULT_CONTEXT,
                   name, selection, &result,&invoke_id));
```

CHECK_DS_CALL is an error-checking macro defined in the example.h header file that is included by example.c. The **ds_read( )** call returns a return code of type DS_status to indicate whether or not the read operation completed successfully. If the call was successful, **ds_read( )** returns the value DS_SUCCESS. If the call fails, it returns an error code. (Refer to “Chapter 10. XDS Class Definitions” on page 189 for a comprehensive list of error codes.) CHECK_DS_CALL interprets this return value and returns successfully to the program or branches to an error-handling routine.

The *session* input parameter is a private object generated by **ds_bind( )** prior to the **ds_read( )** call, as shown in the preceding code fragment. DS_DEFAULT_CONTEXT describes the characteristics of a directory service interaction. Most XDS API function calls require these two input parameters because they define the operating parameters of a session with a CDS server. (Sessions are described in “A Directory Session” on page 125; contexts are described in “The DS_C_CONTEXT Parameter” on page 127.)

The result of a directory service request is returned in a private object (in this case, *result*) that is appropriate to the type of operation. The result of the operation is returned in a single OM object. The components of the result are represented by OM attributes in the operations result object:

- `DS_C_COMPARE_RESULT`

  Returned by **ds_compare( )**
- `DS_C_LIST_RESULT`

  Returned by **ds_list( )**
- `DS_C_READ_RESULT`

  Returned by **ds_read( )**
- `DS_C_SEARCH_RESULT`

  Returned by **ds_search( )**

The OM class returned by **ds_read( )** is `DS_C_READ_RESULT`. The OM class returned by the **ds_compare( )** call is `DS_C_COMPARE_RESULT`, and so on. (Refer to the reference pages in the *IBM DCE Version 3.1 for AIX and Solaris: Application Development Reference* for a description of the OM classes associated with a particular function call; refer to “Chapter 10. XDS Class Definitions” on page 189 for full descriptions of the OM attributes, syntaxes, and values associated with these OM classes.)

The superclasses, subclasses, and OM attributes for `DS_C_READ_RESULT` are shown in Figure 35 on page 135.

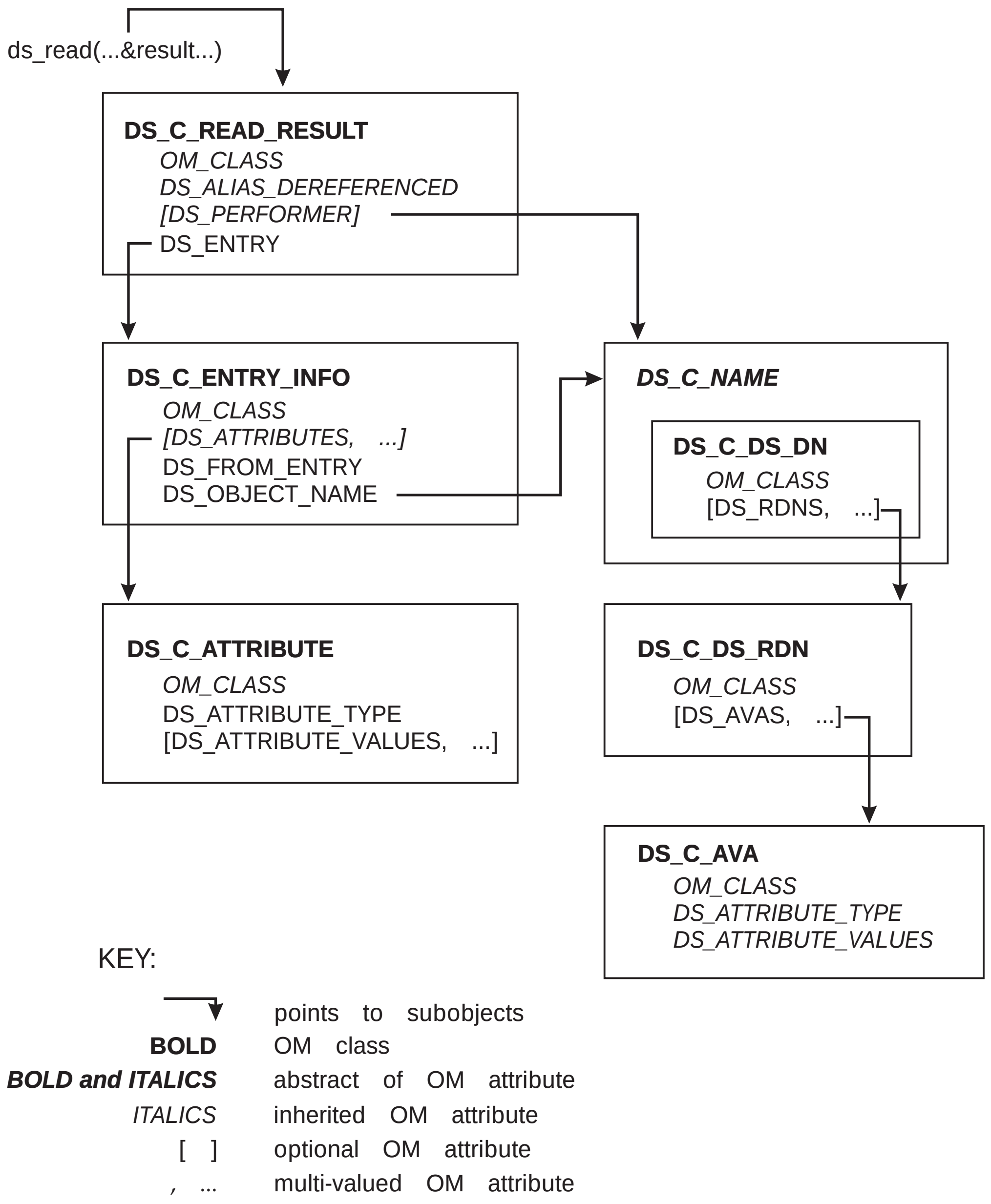


Figure 35. Output from ds_read( ): DS_C_READ_RESULT

The *result* value is returned to the workspace in private implementation-specific format. As such, it cannot be read directly by an application program, but it requires a series of **om_get( )** function calls to extract the requested information from it. The following code fragment from `example.c` shows how a series of **om_get( )** calls extracts the list of telephone numbers associated with the distinguished name for `Peter Piper`. “Chapter 5. XOM Programming” on page 83 describes this extraction process in detail.

```
/*
 * extract the telephone number(s) of "name" from the result
 *
 * There are 4 stages:
 * (1) get the Entry-Information from the Read-Result.
 * (2) get the Attributes from the Entry-Information.
 * (3) get the list of phone numbers.
 * (4) scan the list and print each number.
 */

CHECK_OM_CALL(   om_get( )(result,
             OM_EXCLUDE_ALL_BUT_THESE_TYPES
           + OM_EXCLUDE_SUBOBJECTS,
             entry_list, OM_FALSE, 0, 0, &entry,
             &total_num));

CHECK_OM_CALL(   om_get( )(entry->value.object.object,
             OM_EXCLUDE_ALL_BUT_THESE_TYPES
           + OM_EXCLUDE_SUBOBJECTS,
             attributes_list, OM_FALSE, 0, 0, &attributes,
             &total_num));

CHECK_OM_CALL(   om_get( )(attributes->value.object.object,
             OM_EXCLUDE_ALL_BUT_THESE_TYPES
           + OM_EXCLUDE_SUBOBJECTS,
             telephone_list, OM_FALSE, 0, 0, &telephones,
             &total_num));
```

Directory Search Operations

Search functions can be used to browse through the CDS namespace. For example, a search request could supply the distinguished name of an entry and request a list of the distinguished names of the children of that entry.

XDS over CDS API implements the **ds_list** search operation. With **ds_list**, the requestor supplies a distinguished name. The Directory Service returns a list of the immediate subordinates of the named entry.

Directory Modify Operations

Modify functions alter information in the directory. For example, if an employee of an organizational unit transfers to a new organizational unit, a typical modify request would modify the `OU` name attribute in the person's directory entry to reflect the change.

XDS API implements the following modify functions:

- **ds_modify_entry( )**

  The requestor gives a distinguished name and a list of modifications to the named entry. The XDS/XOM API carries out the specified changes if the user requesting the change has proper access rights.
- **ds_add_entry( )**

  The requestor gives a distinguished name and values for a new entry. The entry is added as a leaf node in the DIT if the user requesting the change has proper access rights. If the leaf entry already exists, it fails and the **ds_modify_entry** must be used.
- **ds_remove_entry( )**

  The requestor gives a distinguished name. The entry with that name is removed if the user requesting the change has proper access rights.

Note: The **ds_add_entry( )** and **ds_remove_entry( )** only apply to leaf entries. They are not intended to provide a general facility for building and manipulating the DIT. Access rights are defined for the session by CDS. The user needs the same access rights as if accessing CDS through **dcecp**.

Modifying Directory Entries

This section describes a modification and subsequent listing of the DIT by using the **ds_add_entry( )**, **ds_list( )**, and **ds_remove_entry( )** function calls. It includes a description of tasks directly related to these operations and does not include service-related tasks. It does not include a **ds_modify_entry( )** function call.

A typical operation to add, remove, or list an entry involves following the same basic steps that were defined previously for the read and search operations:

1. Using the `OM_EXPORT` macro, define the necessary object identifier constants for the OM classes and OM attributes that will define public objects for input to the function calls.
2. Declare the variables that will contain the output from the XDS functions you will use in your application.
3. Build public objects (descriptor lists) for the *name* parameters to the function calls.
4. Create descriptor lists for the attributes to be added, removed, or listed.
5. Perform the operations.

These steps are demonstrated in the following code fragments. The program adds two entries to the directory, then a list operation is performed on their superior entry, and finally the two entries are removed from the directory. The directory tree shown in Figure 36 is used in the program.

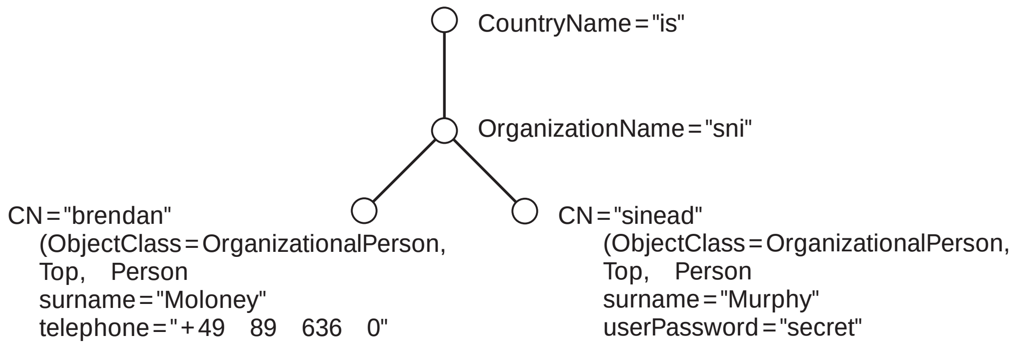


Figure 36. A Sample Directory Tree

Step 1: Export Object Identifiers for Required Directory Classes and Attributes

In the following code fragment, the `OM_EXPORT` macro allocates memory for the constants that represent the object classes and attributes required for the add, list, and remove operations:

```
/* The application has to export the object identifiers   */
/* it requires.     */

OM_EXPORT (DS_C_AVA)
OM_EXPORT (DS_C_DS_RDN)
OM_EXPORT (DS_C_DS_DN)
OM_EXPORT (DS_C_ENTRY_INFO_SELECTION)
OM_EXPORT (DS_C_ATTRIBUTE)
OM_EXPORT (DS_C_ATTRIBUTE_LIST)
OM_EXPORT (DS_A_COUNTRY_NAME)
OM_EXPORT (DS_A_ORG_NAME)
OM_EXPORT (DS_A_ORG_UNIT_NAME)
OM_EXPORT (DS_A_COMMON_NAME)
OM_EXPORT (DS_A_OBJECT_CLASS)
OM_EXPORT (DS_A_PHONE_NBR)
OM_EXPORT (DS_A_USER_PASSWORD)
OM_EXPORT (DS_A_SURNAME)

OM_EXPORT (DS_O_TOP)
OM_EXPORT (DS_O_PERSON)
OM_EXPORT (DS_O_ORG_PERSON)
```

Step 2: Declare Local Variables

The local variables *bound_session*, *result*, and *invoke_id* are defined in the following sample code fragment:

```
OM_private_object bound_session; /* Holds the Session
object  */
                                 /* which is returned by      */
                                 /* ds_bind().                */
OM_private_object result;        /* Holds the list result     */
                                 /* object.                   */
OM_sint           invoke_id;     /* Integer for the invoke id */
                                 /* returned by ds_search().  */
                                 /* This parameter must be    */
                                 /* present even though it is */
                                 /* ignored.
*/
```

These data types are defined in `typedef` statements in the **xom.h** header file. The *bound_session* and *result* variables are defined as data type `OM_private_object` because they are returned by **ds_bind( )** and **ds_list( )** operations to the workspace as private objects. Since asynchronous operations (within the same thread) are not supported, the *invoke_id* parameter functionality is redundant. The *invoke_id* parameter must be supplied to the XDS functions as described in the *IBM DCE Version 3.1 for AIX and Solaris: Application Development Reference*, but its return value should be ignored.

Step 3: Build Public Objects

The public objects required by the **ds_add_entry( )**, **ds_list( )**, and p operations are defined in the following code fragment:

```
/* Build up descriptor lists for the
following distinguished names: */
/*     C=ie/O=sni                                          */
/*     C=ie/O=sni/OU=ap/CN=brendan                         */
/*     C=ie/O=sni/OU=ap/CN=sinead                          */

static OM_descriptor    ava_ie[] = {
    OM_OID_DESC(OM_CLASS, DS_C_AVA),
    OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_COUNTRY_NAME),
    {DS_ATTRIBUTE_VALUES, OM_S_PRINTABLE_STRING, OM_STRING("ie")},
    OM_NULL_DESCRIPTOR
};
static OM_descriptor    ava_sni[] = {
    OM_OID_DESC(OM_CLASS, DS_C_AVA),
    OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_ORG_NAME),
    {DS_ATTRIBUTE_VALUES, OM_S_TELETEX_STRING, OM_STRING("sni")},
    OM_NULL_DESCRIPTOR
};
```

```
static OM_descriptor    ava_ap[] = {
    OM_OID_DESC(OM_CLASS, DS_C_AVA),
    OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_ORG_UNIT_NAME),
    {DS_ATTRIBUTE_VALUES, OM_S_TELETEX_STRING, OM_STRING("ap")},
    OM_NULL_DESCRIPTOR
};
static OM_descriptor    ava_brendan[] = {
    OM_OID_DESC(OM_CLASS, DS_C_AVA),
    OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_COMMON_NAME),
    {DS_ATTRIBUTE_VALUES, OM_S_TELETEX_STRING, OM_STRING("brendan")},
    OM_NULL_DESCRIPTOR
};
static OM_descriptor    ava_sinead[] = {
    OM_OID_DESC(OM_CLASS, DS_C_AVA),
    OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_COMMON_NAME),
    {DS_ATTRIBUTE_VALUES, OM_S_TELETEX_STRING, OM_STRING("sinead")},
    OM_NULL_DESCRIPTOR
};
static OM_descriptor    rdn_ie[] = {
    OM_OID_DESC(OM_CLASS, DS_C_DS_RDN),
    {DS_AVAS, OM_S_OBJECT, {0, ava_ie}},
    OM_NULL_DESCRIPTOR
};
static OM_descriptor    rdn_sni[] = {
    OM_OID_DESC(OM_CLASS, DS_C_DS_RDN),
    {DS_AVAS, OM_S_OBJECT, {0, ava_sni}},
    OM_NULL_DESCRIPTOR
};
static OM_descriptor    rdn_ap[] = {
    OM_OID_DESC(OM_CLASS, DS_C_DS_RDN),
    {DS_AVAS, OM_S_OBJECT, {0, ava_ap}},
    OM_NULL_DESCRIPTOR
};
static OM_descriptor    rdn_brendan[] = {
    OM_OID_DESC(OM_CLASS, DS_C_DS_RDN),
    {DS_AVAS, OM_S_OBJECT, {0, ava_brendan}},
    OM_NULL_DESCRIPTOR
};
static OM_descriptor    rdn_sinead[] = {
    OM_OID_DESC(OM_CLASS, DS_C_DS_RDN),
    {DS_AVAS, OM_S_OBJECT, {0, ava_sinead}},
    OM_NULL_DESCRIPTOR
};
static OM_descriptor dn_ap[] = {
    OM_OID_DESC(OM_CLASS,DS_C_DS_DN),
    {DS_RDNS,OM_S_OBJECT,{0,rdn_ie}},
    {DS_RDNS,OM_S_OBJECT,{0,rdn_sni}},
    {DS_RDNS,OM_S_OBJECT,{0,rdn_ap}},
    OM_NULL_DESCRIPTOR
};
static OM_descriptor dn_brendan[] = {
    OM_OID_DESC(OM_CLASS,DS_C_DS_DN),
    {DS_RDNS,OM_S_OBJECT,{0,rdn_ie}},
    {DS_RDNS,OM_S_OBJECT,{0,rdn_sni}},
    {DS_RDNS,OM_S_OBJECT,{0,rdn_ap}},
    {DS_RDNS,OM_S_OBJECT,{0,rdn_brendan}},
    OM_NULL_DESCRIPTOR
};
static OM_descriptor dn_sinead[] = {
    OM_OID_DESC(OM_CLASS,DS_C_DS_DN),
    {DS_RDNS,OM_S_OBJECT,{0,rdn_ie}},
    {DS_RDNS,OM_S_OBJECT,{0,rdn_sni}},
    {DS_RDNS,OM_S_OBJECT,{0,rdn_ap}},
    {DS_RDNS,OM_S_OBJECT,{0,rdn_sinead}},
    OM_NULL_DESCRIPTOR
};
```

Step 4: Create Descriptor Lists for Attributes

The following code fragments show how the attribute lists are created for the attributes to be added to the directory.

First, initialize the public object `object_class` to contain the representation of the classes in the CDS object that are common to both `Organizational-Person` entries, `Top`, `Person`, and `Organizational-Person`:

```
/* Build up an array of object identifiers for the
 */
/* attributes to be added to the directory.            */

static OM_descriptor    object_class[] = {
 OM_OID_DESC(OM_CLASS, DS_C_ATTRIBUTE),
 OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_OBJECT_CLASS),
 OM_OID_DESC(DS_ATTRIBUTE_VALUES, DS_O_TOP),
 OM_OID_DESC(DS_ATTRIBUTE_VALUES, DS_O_PERSON),
 OM_OID_DESC(DS_ATTRIBUTE_VALUES, DS_O_ORG_PERSON),
 OM_NULL_DESCRIPTOR
};
```

Next, initialize the public objects that represent the attributes to be added. These are `surname` and `telephone` for the distinguished name of Brendan, and `surname2` and `password` for the distinguished name of Sinead:

```
static OM_descriptor    telephone[] = {
 OM_OID_DESC(OM_CLASS, DS_C_ATTRIBUTE),
 OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_PHONE_NBR),
 {DS_ATTRIBUTE_VALUES, OM_S_PRINTABLE_STRING,
 OM_STRING("+49 89 636 0")},
 OM_NULL_DESCRIPTOR
};

static OM_descriptor    surname[] = {
 OM_OID_DESC(OM_CLASS, DS_C_AVA),
 OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_SURNAME),
 {DS_ATTRIBUTE_VALUES, OM_S_TELETEX_STRING,
OM_STRING("Moloney")},
 OM_NULL_DESCRIPTOR
};

static OM_descriptor    surname2[] = {
 OM_OID_DESC(OM_CLASS, DS_C_AVA),
 OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_SURNAME),
 {DS_ATTRIBUTE_VALUES, OM_S_TELETEX_STRING,
 OM_STRING("Murphy")},
 OM_NULL_DESCRIPTOR
};
static OM_descriptor    password[] = {
 OM_OID_DESC(OM_CLASS, DS_C_AVA),
 OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_USER_PASSWORD),
 {DS_ATTRIBUTE_VALUES, OM_S_OCTET_STRING,
OM_STRING("secret")},
 OM_NULL_DESCRIPTOR
};
```

Finally, initialize the public objects that represent the list of attributes to be added to the directory. These are *attr_list1* for the distinguished name Brendan, and *attr_list2* for the distinguished name Sinead:

```
static OM_descriptor    attr_list1[] = {
 OM_OID_DESC(OM_CLASS, DS_C_ATTRIBUTE_LIST),
 {DS_ATTRIBUTES, OM_S_OBJECT, {0, object_class} },
 {DS_ATTRIBUTES, OM_S_OBJECT, {0, surname} },
 {DS_ATTRIBUTES, OM_S_OBJECT, {0, telephone} },
 OM_NULL_DESCRIPTOR
};

static OM_descriptor    attr_list2[] = {
 OM_OID_DESC(OM_CLASS, DS_C_ATTRIBUTE_LIST),
 {DS_ATTRIBUTES, OM_S_OBJECT, {0, object_class} },
 {DS_ATTRIBUTES, OM_S_OBJECT, {0, surname2} },
 {DS_ATTRIBUTES, OM_S_OBJECT, {0, password} },
 OM_NULL_DESCRIPTOR
};
```

The *attr_list1* variable contains the public objects `surname` and `telephone`, which are the C representations of the attributes of the distinguished name `/C=ie/O=sni/OU=ap/CN=Brendan` that are added to the directory. The *attr_list2* variable contains the public objects first `surname2` and `password`, which are the C representations of the attributes of the distinguished name `/C=ie/O=sni/OU=ap/CN=Sinead`.

Step 5: Perform the Operations

The following code fragments show the **ds_add_entry( )**, **ds_list( )**, and the **ds_remove_entry( )** calls.

First, the two **ds_add_entry( )** function calls add the attribute lists contained in *attr_list1* and *attr_list2* to the distinguished names represented by `dn_brendan` and `dn_sinead`, respectively:

```
/* Add two entries to the GDS server.               */

if (ds_add_entry(bound_session, DS_DEFAULT_CONTEXT,
   dn_brendan, attr_list1,
   &invoke_id) != DS_SUCCESS)
   printf("ds_add_entry() error\n");

if (ds_add_entry(bound_session, DS_DEFAULT_CONTEXT,
   dn_sinead, attr_list2,
   &invoke_id) != DS_SUCCESS)
   printf("ds_add_entry() error\n");
```

Next, list all the subordinates of the object referenced by the distinguished name `/C=ie/O=sni/OU=ap`:

```
if (ds_list(bound_session, DS_DEFAULT_CONTEXT, dn_ap,
   &result, &invoke_id)
   != DS_SUCCESS)
   printf("ds_list() error\n");
```

The **ds_list( )** call returns the result in the private object `result` to the workspace. The components of `result` are represented by OM attributes in the OM class `DS_C_LIST_RESULT` (as shown in Figure 37 on page 142) and can only be read by a series of **om_get( )** calls.

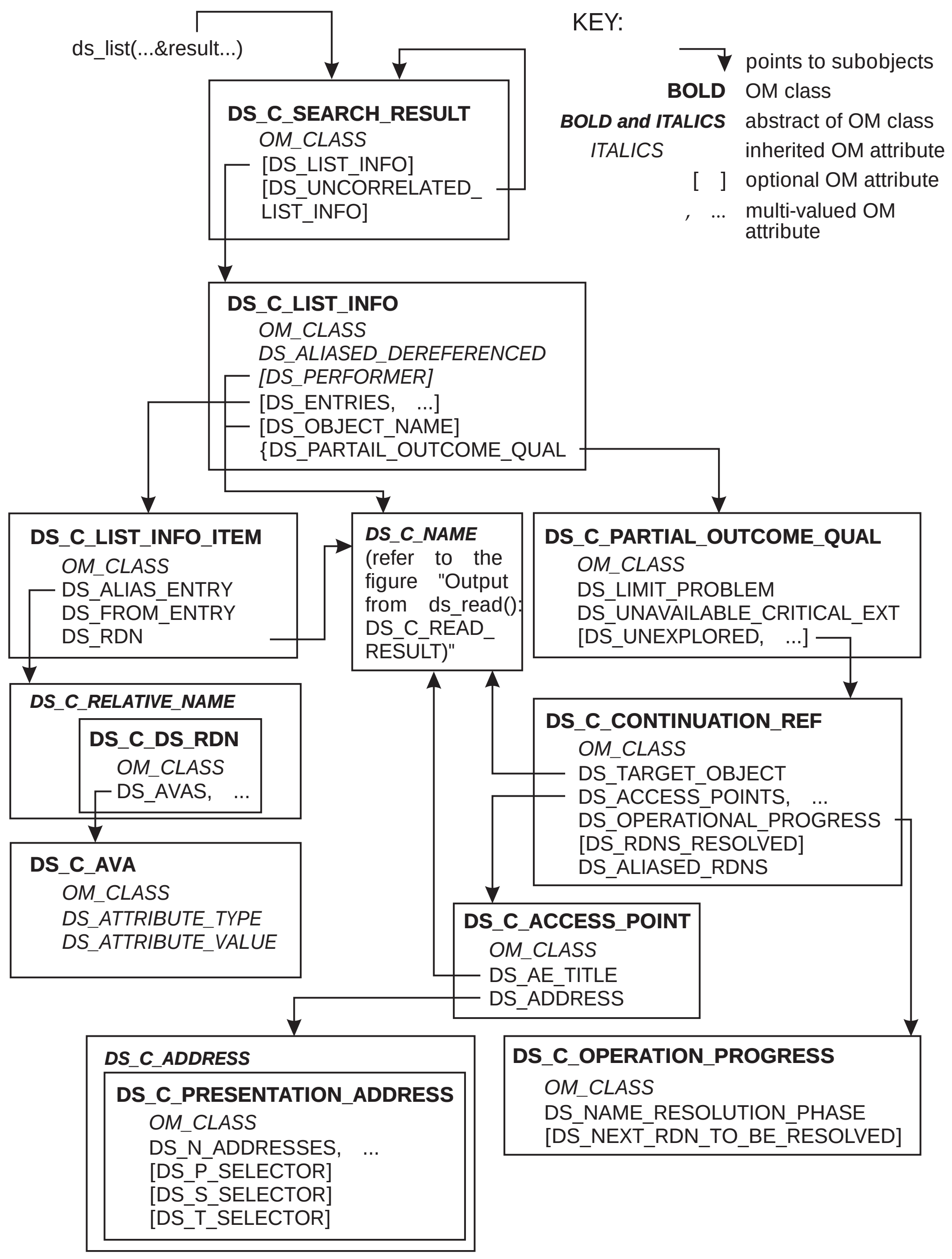


Figure 37. OM Class DS_C_LIST_RESULT

Finally, remove the two entries from the directory:

```
if (ds_remove_entry(bound_session, DS_DEFAULT_CONTEXT,
    dn_brendan, &invoke_id)
    != DS_SUCCESS)
    printf("ds_remove_entry() error\n");

if (ds_remove_entry(bound_session, DS_DEFAULT_CONTEXT,
    dn_sinead, &invoke_id)
    != DS_SUCCESS)
    printf("ds_remove_entry() error\n");
```

Return Codes

XDS API function calls return a value of type `DS_status`, with the exception of **ds_initialize( )** which returns a value of type `OM_workspace`. If the function is successful, then `DS_status` returns with a value of `DS_SUCCESS`. If the function does not complete successfully, then `DS_status` takes either the error constant `DS_NO_WORKSPACE` or one of the private error objects described in “Chapter 10. XDS Class Definitions” on page 189.

Chapter 7. Using Threads With The XDS/XOM API

Some programs work well when they are structured as multiple flows of control. Other programs may show better performance when they are multithreaded, allowing the multiple threads to be mapped to multiple processors when they are available.

XDS/XOM application programs can contain multiple threads of control. For example, a XDS/XOM application may need to query several CDS servers. This can be achieved more efficiently by using separate threads simultaneously to query the different servers.

XDS/XOM supports multithreaded applications. Writing multithreaded applications over XDS/XOM imposes new requirements on programmers. They must manage the threads, synchronize threads' access to global resources, and make choices about thread scheduling and priorities.

This chapter describes a simple XDS/XOM application that uses threads. (Refer to the `*(3thr)` reference pages for more information on DCE threads.)

The XDS/XOM API calls do not change when they are making use of DCE threads in an application program. The service underneath XDS/XOM API is designed to be both *thread-safe*, to allow multiple threads to safely access shared data, and *cancel-safe*, to handle unexpected cancellation of a thread in an application program.

Figure 38 on page 146 shows an example of how an application can issue XDS/XOM calls from within different threads.

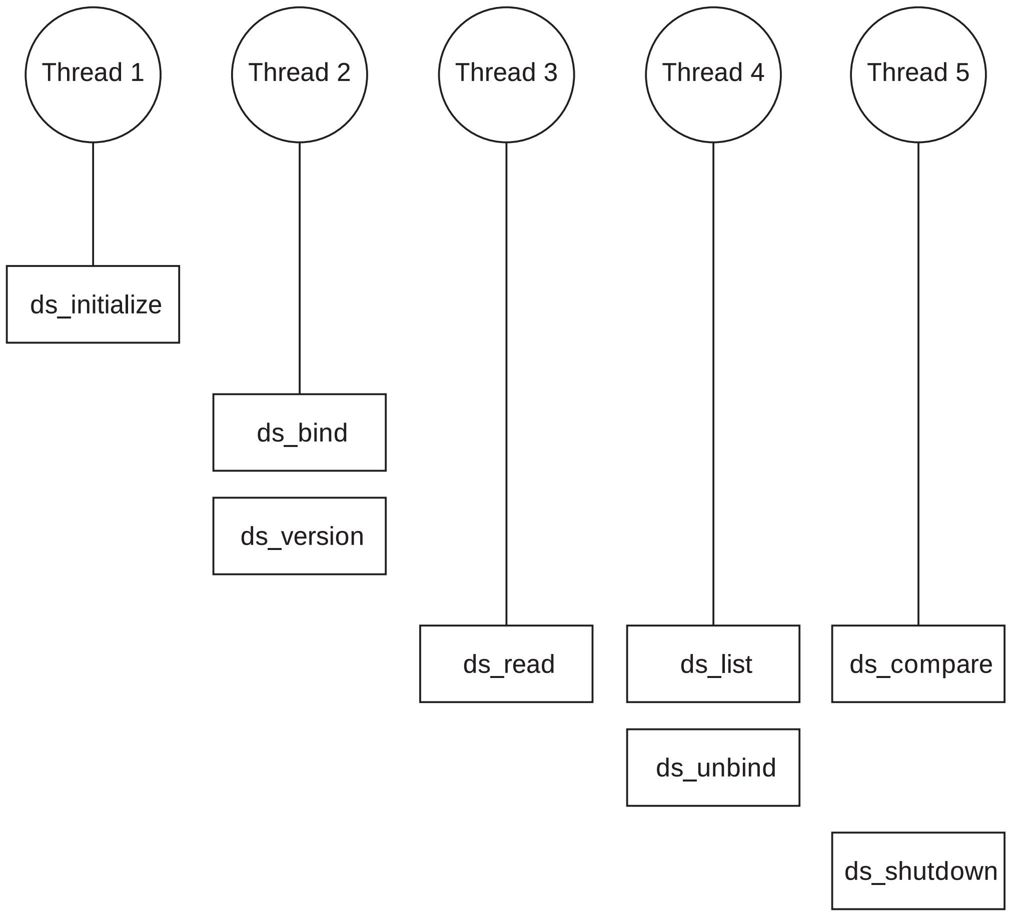


Figure 38. Issuing XDS/XOM Calls from Within Different Threads

The order of thread completion is not defined; however, XDS/XOM has an inherent ordering. Multithreaded XDS applications must adhere to the following order of execution:

1. **ds_initialize( )**
2. **ds_version( )** (optional)
3. **ds_bind( )**
4. Other XDS calls in sequence or parallel from multiple threads
5. **ds_unbind( )**
6. **ds_shutdown( )**

Multithreaded XOM applications must adhere to the following order of execution:

1. **ds_initialize( )**
2. XOM calls in sequence or parallel from multiple threads
3. **ds_shutdown( )**

The XDS/XOM API returns an appropriate error code if these sequences are not adhered to. For example the following errors are returned:

DS_E_BUSY
If **ds_unbind(**) is called while there are still outstanding operations, or if **ds_shutdown( )** is called before all directory connections have been released by **ds_unbind( )**.

OM_NO_SUCH_WORKSPACE
: If any XOM API calls are made before calling **ds_initialize( )**, or if a call to **ds_shutdown( )** completes while there are outstanding XOM operations on the same workspace. In the latter case, these XOM operations will not be performed.

Overview of Sample Threads Program

The sample program is called `thradd`. The `thradd` program is a multithreaded XDS application that adds entries to a CDS directory. Each thread performs a **ds_add_entry( )** call. The information for each entry to be added is read from an input file.

The `thradd` program can also be used to reset the directory to its original state. This is achieved by invoking `thradd` with a `-d` command-line argument. In this case, `thradd` uses the same input file and calls **ds_remove_entry( )** for each entry. The **ds_remove_entry( )** calls are also done in separate threads.

To keep the program short and clear, it works with a fixed tree for the upper nodes (`/C=it/O=sni/OU=ap`), to which the entries described in the input file are added. This fixed upper tree is added to the directory by `thradd`. The input file contains the common name, the surname, and the phone number of each `Organizational-Person` entry to be added.

For simplicity, only **pthread_join( )** is used for synchronization purposes; mutexes are not used.

The `thradd` program can be enhanced to satisfy the following scenarios:

- As a server program for interactive directory actions from different users. The `thradd` program simulates a server program that gets requests from different users to add entries to a directory. In the case of `thradd`, the users' interactive input is simulated through the entries in the input file. Each line of input represents a different directory entry, and `thradd` uses a separate thread for each line.
- Initialization of the directory with data from file. The `thradd` program could be enhanced to read generic attribute information for a variety of directory object classes from a file, and to add the corresponding entries to the directory.

User Interface

The `thradd` program is called from the command line as follows:

thradd [**-d**] [**-f** *file_name*]

where:

-d
: Causes the entries in the file and the tree `/C=it/O=sni/OU=ap` to be deleted; otherwise, they are added.

-f *file_name*
: Specifies the name of the input file. If no input file is specified, then a default filename of **thradd.dat** is used.

Input File Format

The input file can contain any number of lines. Each line represents a directory entry of an organizational person. Each line must contain the following three attributes for each entry:

```
<common name>   <surname>   <phone number>
```

The attributes must be strings without space characters. Lines containing less than three strings are rejected by the program; any input on a line after the first three strings is ignored and can be used for comments. The attributes are separated by one or more space characters.

The input strings are not verified for their relevant attribute syntax. A wrong attribute syntax will result in either a **ds_add_entry( )** error or a **ds_remove_entry( )** error.

The following would be a valid input file for `thradd`:

```
Anna          Meister        010101
Erwin         Reiter         020202
Gerhard       Schulz         030303
Gottfried     Schmid         040404
Heidrun       Blum           050505
Hermann       Meier          060606
Josefa        Fischer        070707
Jutta         Arndt          080808
Leopold       Huber          090909
Magdalena     Schuster       101010
Margot        Junge          111111
```

Program Output

The `thradd` program writes messages to `stdout` for every action done by a thread. The order of the output can differ from the order in the input file; it depends on the execution of the different threads.

Errors are reported to `stderr`.

Prerequisites

The directory must be active before running `thradd`. If you are running `thradd` in *adding* mode then the directory should not contain a node `/C=it`. The `thradd` program should always be invoked twice with the same input file: first without and then with option `-d`. This guarantees that the directory is reset to its original state. The DCE administration program **dcecp** can be used to verify the directory contents after adding entries.

Description of Sample Program

The `thradd` program has a similar structure to the sample XDS programs in the previous chapter. Therefore, only a short general outline of the program is given here. The thread specifics are described in detail in the next section.

The static descriptors for the fixed tree (that is, `/C=it/O=sni/OU=ap`) are declared in the **thradd.h** header file. The **thradd.c** application and the **thradd.h** header file are shipped with the product and are located in the **/opt/dcelocal/examples/xdsxom** directory.

The main routine scans the command-line options, initializes the XDS workspace and binds to the CDS namespace.

The program then binds to the default CDS server. Each line of the input file is processed in turn by a `while` loop (until the end of the file is reached). The `while` loop contains two `for` loops. The first `for` loop creates a separate thread for each line of the input file, up to a maximum of `MAX_THREAD_NO` of threads.

The `add_or_remove( )` procedure, which adds or removes an entry to/from the directory, is the starting point of each thread's processing.

The second `for` loop waits for termination of the threads and then releases the resources used by the threads.

When the entire input file has been processed, `thradd` closes the connection to the CDS server and, if working in *removing* mode, removes the fixed tree of upper nodes (that is, `/C=it/O=sni/OU=ap`).

Finally, the XDS workspace is closed.

Figure 39 on page 150 shows the program flow.

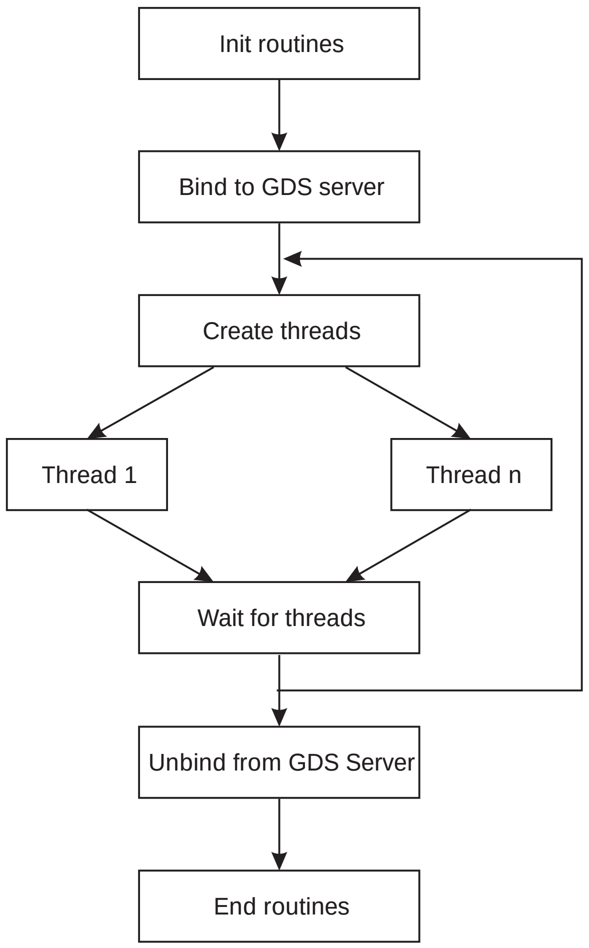


Figure 39. Program Flow for the thradd Sample Program

Detailed Description of Thread Specifics

The program consists of the following general steps:

1. Include the header file **pthread.h**.
2. Define a parameter block structure type for the thread start routine.
3. Declare arrays for thread handles and parameter blocks.
4. Read the input file line by line.
5. Update the parameter block.
6. Create the thread.
7. Wait for the termination of the thread.
8. Release the resources used by the thread.
9. Define the thread start routine.
10. Declare local variables needed for descriptors for the objects read from the input file.

The following paragraphs describe the corresponding step numbers from the program listing in the next section:

Step 1 includes the header file **pthread.h**, which is required for thread programming.

Step 2 defines a parameter block structure type for the thread start routine. A thread start routine must have exactly one parameter. However, **add_or_remove( )** requires three parameters (session object, input line, and operating mode). The structure `pb_add_or_remove` is defined as the parameter block for these components. Therefore, the single parameter block contains the three parameters required by `add_or_remove( )`.

Step 3 declares arrays for thread handles and parameter blocks. The routine that creates the thread (`main`, in this case) must maintain the following information for each thread:

- A thread handle of type `pthread_t` to identify the thread for join and detach calls.
- A thread-specific parameter block that cannot be accessed by any other thread. This makes sure that a parameter for one thread is not overwritten by another thread.

Step 4 reads the input file line by line. A thread is created for each line. A maximum `MAX_THREAD_NO` of threads is created in parallel. The program then waits for the termination of the created threads so that it can release the resources used by these threads, allowing it to create new threads for remaining input lines (if any).

The absolute maximum number of threads working in parallel depends on system limits; for `thradd`, a value of 10 was chosen (see **thradd.h**), which is well below the maximum on most systems.

Step 5 updates the parameter block. For each thread, a different element of the array of parameter blocks is used.

Step 6 creates the thread. The thread is created by using the function **pthread_create( )**. The function has the following parameters:

- The thread handle (output) is stored in an element of the array of type `pthread_t`.
- For the thread characteristics, the default `pthread_attr_default` is used.
- The start routine for this thread is `add_or_remove( )`.
- The parameter passed to `add_or_remove( )` is a pointer to an element of the array of parameter blocks.

Step 7 waits for the termination of the thread. The pthread_join( ) routine is called with the thread handle as the input parameter. The program waits for the termination of the thread. If the thread has already terminated, then **pthread_join( )** returns immediately. The second parameter of **pthread_join( )** contains the return value of the start function; here it is a dummy value because **add_or_remove( )** returns a `void`. The **add_or_remove( )** routine is designed as a `void` function because the calling routine does not have to deal with error cases. The **add_or_remove( )** routine prints status messages itself to show the processing order of the threads. Usually, a status should be returned to the application.

Step 8 releases the resources used by the thread. The thread handle is used as input for the function **pthread_detach( )**, which releases the resources (for example, memory) used by the thread.

Step 9 defines the thread start routine. As previously mentioned, the thread start routine must have exactly one parameter. In this case, it is a pointer to the parameter block structure defined in Step 2.

Step 10 declares local variables needed for descriptors for the objects read from the input file. These descriptors are variables and are declared as automatic because of the reentrancy requirement. In the previous sample programs, descriptors were generally declared static. For this example, this is only possible for the constant descriptors declared in **thradd.h**.

Of course, this example shows only a small part of the possibilities of multithreaded XDS programming.

Chapter 8. XDS/XOM Convenience Routines

This chapter describes functions that are available to XDS/XOM programmers to help simplify and speed up the development of XDS applications. The convenience functions target two main areas, as follows:

- Filling, comparing, and extracting objects
- Converting objects to and from strings

The following six convenience functions are provided:

- **dsX_extract_attr_values( )**
- **omX_fill( )**
- **omX_fill_oid( )**
- **omX_extract( )**
- **omX_string_to_object( )**
- **omX_object_to_string( )**

Refer to the `*(3xds)` and `*(3xom)` reference pages for detailed descriptions of these functions.

To demonstrate the power of the convenience functions, the **acl.c** sample program located in **/opt/dcelocal/examples/xdsxom** is presented again here, after being modified to make use of these functions. The modified sample program is called **acl2.c**.

String Handling

The convenience functions provide the ability to specify OM objects in string format by means of abbreviations. These abbreviations are defined in the XOM object information file `xoischema`.

X.500 attribute types can be specified as abbreviations or object identifier strings. The mapping of the attribute abbreviations and object identifier strings to BER encoded object identifiers and the associated attribute syntaxes is determined by the XOM object information module with the help of the `xoischema` file. For valid attribute abbreviations, please refer to the `xoischema` file in the following directory:

```
dce_local_path>/var/adm/directory/gds/adm
```

It is important that any schema changes to the DSA are reflected in the `xoischema` file.

The convenience functions are able to handle strings with special syntax. The strings can be broadly classified into the following:

- Strings representing GDS attribute information
- Strings representing structured GDS attribute information
- Strings representing a structured GDS attribute value
- Strings representing a distinguished name (DN)
- Strings representing expressions

Strings Representing GDS Attribute Information

Strings that represent GDS attribute information are used to associate the attributes with their values. They are of the form:

```
attribute_type =
attribute_value
```

The attribute types can either be specified as abbreviations or object identifier strings. An object identifier string is defined as a series of digits separated by the **.** (dot) character. If attribute abbreviations are used, they are case insensitive. For example, `cn=schmid` or `85.4.3=schmid`.

In the case of attributes with `OM_S_OBJECT_IDENTIFIER` syntax, the attribute value can also be specified as an abbreviation string. For example, an object class for `Residential Person` can be specified as `OCL=REP` or `OCL='\x55\x06\x0A'`

All leading and trailing whitespace (surrounding the attribute type, the = (equal sign), and the attribute value) is ignored.

The following are the reserved characters for such strings:

'	Used to enclose the attribute values. If this character is used, all other reserved characters within the quoted string except the \ (backslash) are not interpreted. For example, `cn=henry mueller`
;	Separates multiple values of a recurring attribute. All leading and trailing whitespace (surrounding the semicolon) is ignored. For example, `TN=899898;979779`
=	Associates the attribute with its value.
\x *nn*	Specifies hexadecimal data. The two characters *nn* are read as the hexadecimal value.
\	Used to escape any of the other reserved characters.

Strings Representing Structured GDS Attribute Information

Strings that represent structured GDS attribute information are used to associate the structured attribute and its components with their values. They are of the form:

```
structured_attribute_type = {
Comp1 =
Value,
Comp2 =
Value, ..}
```

The structured attribute type can be specified as abbreviations or object identifier strings. An object identifier string is defined as a series of digits separated by dots. If attribute abbreviations are used, they are case insensitive. *Comp1*, *Comp2*, and so on, are the components of the structured attribute. They should be specified as abbreviations, as in the following example:

```
TXN={TN=977999, CC=345, AB=8444}
```

Recurring values for structured attributes can be specified with the help of the semicolon. An example follows:

```
TXN={TN=977999, CC=345, AB=8444};{TN=123444,CC=345,
AB=8444}
```

Recurring values for the components should be specified as follows:

```
TXN={TN=977999; 274424, CC=345, AB=8444}
```

If any of the components are further structured, they should be enclosed within braces as follows:

```
FTN={PA={FR=1,TD=1}, PN=67899}
```

All leading and trailing whitespace, which surrounds the structured attribute type, the component abbreviation, the equal sign, the { (left brace), the **,** (comma), and the } (right brace), is ignored.

Attributes and components with DN syntax should be specified as follows:

```
AON={/c=de/o=sni/ou=ap11/cn=mueller}
ACL={MPUB={INT=0, USR={/c=de/o=sni/cn=mueller,
sn=schmid}}}
```

In the case of attributes with `OM_S_OBJECT_IDENTIFIER` syntax, the attribute value can also be specified as an abbreviation string, as shown in the following:

```
SG={OCL=REP}
SG={OCL='\x55\x06\x0A'}
```

Attributes of type presentation address (OM class `DS_C_PRESENTATION_ADDRESS`) are handled specially, using the PSAP macro utility. The value for such an attribute can be specified as follows:

```
PSA={TS=Server,
NA='TCP/IP!internet=127.0.0.1+port=12345'}
```

The *local_string* parameter should be set to `OM_TRUE` in the convenience function being used. Here, the network address (NA) is specified with a special syntax. Refer to the *OSF DCE GDS Administration Guide and Reference* for further information.

The following are the reserved characters for strings with structured attribute information:

' Used to enclose the attribute values. If this character is used, all other reserved characters within the quoted string except the backslash are not interpreted. For example, `cn='henry mueller'`

/ Specifies an attribute value with DN syntax. For example, `AON = {/c=de/o=sni/ou=ap22/cn=mayer}`

{ Indicates the start of a structured attribute value block.

} Indicates the end of a structured attribute value block.

, Separates the components of a structured attribute. For example, `TN=977999, CC=345, AB=8444`

It can also be used to specify multiple AVAs in the case of attributes with DN syntax.

; Separates multiple values of a recurring attribute or the recurring components of the structured attribute. All leading and trailing whitespace (surrounding the attribute type, the equal sign, the left and right braces, the component abbreviation, the component value and the semicolon) is ignored. The following is an example:

```
TXN={TN=977999,CC=345,AB=8444};{TN=53533,CC=242,AB=44242}
```

= Associates the components with their values, and associates the components to the structured attribute.

\x *nn* Used to specify hexadecimal data. The two characters *nn* are read as the hexadecimal value.

\ Used to escape any of the other reserved characters.

Strings Representing a Structured GDS Attribute Value

Strings are used to represent the structured GDS attribute value. Only one structured attribute value can be specified.

They are of the form:

```
Comp1 = Value,
Comp2 = Value, ....
```

Comp1, *Comp2*, and so on, are the components of the structured attribute. They should be specified as abbreviations. For example, to specify a value for `DS_C_TELEX_NBR` class, the string format is the following:

```
TN=977999, CC=345, AB=8444
```

Recurring values for the components can be specified as shown in the following:

```
TN=977999; 274424, CC=345, AB=8444
```

If any of the components are further structured, they should be enclosed within braces as follows:

```
FTP={FR=1,TD=1}, PN=67899
```

Components with DN syntax can be specified as follows:

```
MPUB={INT=0, USR={/c=de/o=sni/cn=mueller,
sn=schmid}}
```

Components of type presentation address (OM class `DS_C_PRESENTATION_ADDRESS`) are handled specially, using the PSAP macro utility. The value for the components can be specified as follows:

```
TS=Server,
NA='TCP/IP!internet=127.0.0.1+port=12345'
```

The *local_string* parameter should be set to `OM_TRUE` in the convenience function being used. Here, the NA is specified with a special syntax. Refer to the *OSF DCE GDS Administration Guide and Reference* for further information.

The reserved characters for such strings are the same as those for strings representing structured attribute information ("Strings Representing Structured GDS Attribute Information" on page 154).

Strings Representing a Distinguished Name

Strings are used to represent the DN of the object. They are of the form:

```
/
attribute_type =
naming_attribute_value ....
```

or

```
/
attribute_value/
attribute_value ....
```

The attribute types can be specified as abbreviations or object identifier strings. An object identifier string is defined as a series of digits separated by dots. If attribute abbreviations are used, they are case insensitive. Multiple AVAs are represented by separating the naming attribute values with commas.

The first RDN can also be specified as the DCE global root string **/...**, which is a sequence of the slash followed by three dots. In this case, the **/...** string is simply ignored and the rest of the string is processed. Three examples follow:

```
/c=de/o=sni/ou=ap11, l=munich/85.4.3=schmid
/c=us/o=osf/ou=abc/subsystems/server/xyz
/.../c=us/o=osf/ou=abc/subsystems/server/xyz
```

The first nonspace character should always be the slash. All leading and trailing whitespace (surrounding the slash, the attribute type, the equal sign and the attribute value) is ignored.

The following are the reserved characters:

'	Used to enclose the naming attribute values. If this character is used, all other reserved characters within the quoted string except the backslash are not interpreted. For example, `cn='henry mueller'`.
/	Used as a delimiter between RDNs.
,	Specifies multiple AVAs. All leading and trailing whitespace surrounding the comma is ignored. An example follows: `/c=de/o=dbp/ou=dap11/cn=schmid, ou=ap11`
=	Associates the object with its naming attribute value.
\x *nn*	Used to specify hexadecimal data. The two characters *nn* are read as the hexadecimal value.
\	Used to escape any of the other reserved characters.

Strings Representing Expressions

Strings are used to specify an SQL-like expression in a search operation. For example, consider the following:

```
(CN ˜ =schmid) && (OCL=ORP || OCL=REP) && !(SN=ronnie)
```

This is used to search for anybody who is an organizational person or a residential person, whose name approximately matches `schmid` but whose surname is not `ronnie`.

Object identifiers can also be used instead of attribute abbreviations. The object identifier string is a series of numbers separated by dots.

All leading and trailing whitespace (surrounding the attribute types, the operators, and the attribute values) is ignored.

If spaces are part of the attribute value, then the complete attribute value must be enclosed in quotes.

Additionally, the presence of an attribute can also be tested in either of the following ways:

```
c = de && cn
c = de && cn = *
```

The following are the reserved characters:

' Used to indicate the start/end of an attribute value string. Can be used when spaces are part of the data. If this character is used, all other reserved characters within the quoted string except the backslash are not interpreted. An example follows:

```
OU=sni && cn='Henri Mueller' &&tn=89989
```

/ Used to specify an attribute value with DN syntax. An example follows:

```
AON ={/c=de/o=sni/ou=ap22/cn=mayer}
```

= Used to associate the attribute with its value.

&& Used to logically AND two conditions.

|| Used to logically OR two conditions.

! Used to logically NEGATE a condition.

˜= Used to specify phonetic matching during a search operation.

> Used to match values greater than a specified value.

>= Used to match values greater than or equal to a specified value.

< Used to match values less than a specified value.

<= Used to match values less than or equal to a specified value.

***** Used to specify substrings during search.

(Used for nesting of filters.

) Used for nesting of filters.

{	Indicates the start of a structured attribute value block.
}	Indicates the end of a structured attribute value block.
,	Separates the components of a structured attribute. For example, `TN=977999`**,** `CC=345`**,** `AB=8444` It can also be used to specify multiple AVAs in the case of attributes with DN syntax.
\x *nn*	Used to specify hexadecimal data. The two characters *nn* are read as the hexadecimal value.
\	Used to escape any of the other reserved characters.

During evaluation of complex expressions during search operations, the following precedence of operators prevail:

1. `( )`
2. `!`
3. `&&`
4. `||`

The ( ) operators have the highest precedence, and **||** the lowest.

The acl2.c Program

The **acl2.c** file is a program that performs the same functionality as **acl.c** in the sample files located in the **/opt/dcelocal/examples/xdsxom** directory. The purpose of **acl2.c** and **acl2.h** is to show how the XDS/XOM convenience functions can be used to reduce the complexity of a real application.

The program consists of the following steps:

1. Export the required object identifiers. (See the **acl2.h** description in "The acl2.h Header File" on page 174.)
2. Define the string expressions for the directory entry names and their attributes. (See the **acl2.h** description in "The acl2.h Header File" on page 174.)
3. Initialize a workspace.
4. Negotiate use of the basic directory contents and GDS packages.
5. Build the name objects for the entries to be added to the directory.
6. Build the attribute objects for the entries to be added to the directory.
7. Add the fixed tree of entries to the directory in order to permit an authenticated bind.
8. Create a default session object.
9. Alter the default session object to include the credentials of the requestor (`/C=de/O=sni/OU=ap/CN=norbert`).
10. Bind with credentials to the default GDS server.
11. Create a default context object and alter it to include shadow entries.
12. Build filter, name, and entry information selection objects to be used for the search process.
13. Search the whole subtree below `root` and extract the ACL attribute from each selected entry.
14. Close the connection to the GDS server.
15. Remove the user's credentials from the directory.
16. Release the memory used for application-created objects.

17. Extract the components from the search result.
18. Examine each entry and print the entry details.
19. Close the XDS workspace.

In comparison to the **acl.c** program located in **/opt/dcelocal/examples/xdsxom**, the following points should be noted:

1. Step 1 has not changed significantly. The number of object identifiers, which the **acl2.c** needs to be exported, has been reduced.
2. Step 2 has been completely revised. In fact, the header file has been reduced substantially. This is as a result of removing all the static descriptor lists for the directory names and attributes and replacing them with string expressions.
3. Steps 3 and 4 are the same as before.
4. Steps 5 and 6 are new steps that make use of the convenience functions **omX_string_to_object(**), **omX_fill_oid(**), and **omX_fill( )**.
5. Steps 7 through 10 are the same as Steps 5 through 8.
6. Step 11 is the same as Step 9, but with an additional call to build an object to specify the use of shadow entries. A convenience function is used for this purpose. This replaces a static descriptor list definition from the old header file.
7. Step 12 is new. It calls several convenience functions to create objects that are used by **ds_search( )**. These objects were statically declared in the header file.
8. Steps 13 through 15 are the same as Steps 10 through 12 from the old code.
9. Step 16 is a new step to release memory that has been allocated by the convenience functions when creating objects.
10. Step 17 replaces Step 13 from the old program with a call to the convenience function **omX_extract( )** to extract the required components from the search result.
11. Step 18 is the same as Step 14 in the old program, but with an additional call to free the memory allocated by **omX_extract( )** in the previous step.
12. Step 19 is the same as Step 15 in the old code.

The acl2.c Code

The following code is a listing of the **acl2.c** program:

```
/***************************************************************
*                                                              *
*  COPYRIGHT (C) SIEMENS NIXDORF INFORMATIONSSYSTEME AG 1991   *
*               ALL RIGHTS RESERVED                            *
*                                                              *
***************************************************************/

/*
 * This sample program displays the access permissions (ACL) on each
 * entry in the directory for a specific user.  The permissions are
 * presented in a form similar to the UNIX file permissions.  In
 * addition, each entry is flagged as either a master or a shadow copy.
 *
 * The distinguished name of the user performing the check is:
 *
 *     /C=de/O=sni/OU=ap/CN=norbert
 *
 * The results are presented in the following format:
 *
 *     [ABCD] <entry's distinguished name>
 *
```

```
*     A:  'm' master copy
*         's' shadow copy
*
*     B:  'r' read access to public attributes
*         'w' write access to public attributes
*         '-' no access to public attributes
*
*     C:  'r' read access to standard attributes
*         'w' write access to standard attributes
*         '-' no access to standard attributes
*
*     D:  'r' read access to sensitive attributes
*         'w' write access to sensitive attributes
*         '-' no access to sensitive attributes
*
* For example, the following result means that the entry
* '/C=de/O=sni' is a master copy and that the requesting user
* (/C=de/O=sni/OU=ap/CN=norbert) has write access to its public
* attributes, read access to its standard attributes and no access
* to its sensitive attributes.
*
*      [mwr-] /C=de/O=sni
*
* The program requires that the specific user perform an authenticated
* bind to the directory.  In order to achieve this the user's
* credentials must already exist in the directory.  Therefore the
* following tree of 6 entries is added to the directory each time the
* program runs, and removed again afterwards.
*
*             O  C=de
*             |  (objectClass=Country,
*             |   ACL=(mod-pub: *
*             |        read-std:*
*             |        mod-std: *
*             |        read-sen:*
*             |        mod-sen: *))
*             |
*             |
*             O  O=sni
*             |  (objectClass=Organization,
*             |   ACL=(mod-pub: /C=de/O=sni/OU=ap/*
*             |        read-std:/C=de/O=sni/OU=ap/CN=stefanie
*             |        mod-std: /C=de/O=sni/OU=ap/CN=stefanie
*             |        read-sen:/C=de/O=sni/OU=ap/CN=stefanie
*             |        mod-sen: /C=de/O=sni/OU=ap/CN=stefanie))
*             |
*             O  OU=ap
*             |  (objectClass=OrganizationalUnit,
*             |   ACL=(mod-pub: /C=de/O=sni/OU=ap/*
*             |        read-std:/C=de/O=sni/OU=ap/CN=stefanie
*             |        mod-std: /C=de/O=sni/OU=ap/CN=stefanie
*             |        read-sen:/C=de/O=sni/OU=ap/CN=stefanie
*             |        mod-sen: /C=de/O=sni/OU=ap/CN=stefanie))
*             |
*      +-------+-------+
*      |       |       |
*      |       |       O  CN=ingrid
*      |       |         (objectClass=OrganizationalPerson,
*      |       |          ACL=(mod-pub: /C=de/O=sni/OU=ap/*
*      |       |               read-std:/C=de/O=sni/OU=ap/*
*      |       |               mod-std: /C=de/O=sni/OU=ap/CN=stefanie
*      |       |               read-sen:/C=de/O=sni/OU=ap/*
*      |       |               mod-sen: /C=de/O=sni/OU=ap/CN=stefanie),
*      |       |          surname="Schmid",
*      |       |          telephone="+49 89 636 0",
*      |       |          userPassword="secret")
*      |       |
 *      |       O  CN=norbert
 *      |          (objectClass=OrganizationalPerson,
 *      |           ACL=(mod-pub: /C=de/O=sni/OU=ap/*
 *      |                read-std:/C=de/O=sni/OU=ap/*
 *      |                mod-std: /C=de/O=sni/OU=ap/CN=stefanie
 *      |                read-sen:/C=de/O=sni/OU=ap/*
 *      |                mod-sen: /C=de/O=sni/OU=ap/CN=stefanie),
 *      |           surname="Schmid",
 *      |           telephone="+49 89 636 0",
 *      |           userPassword="secret")
 *      |
 *      O  CN=stefanie
 *         (objectClass=OrganizationalPerson,
 *          ACL=(mod-pub: /C=de/O=sni/OU=ap/*
 *               read-std:/C=de/O=sni/OU=ap/*
 *               mod-std: /C=de/O=sni/OU=ap/CN=stefanie
 *               read-sen:/C=de/O=sni/OU=ap/*
 *               mod-sen: /C=de/O=sni/OU=ap/CN=stefanie),
 *          surname="Schmid",
 *          telephone="+49 89 636 0",
 *          userPassword="secret")
 *
 *
 * In this version of the program, instead of providing client-generated

 * public objects, the XOM Convenience Functions are used for creating
 * objects.  They are also used for extracting information from service
 * generated objects.
 */

#ifdef THREADSAFE
#include <pthread.h>
#endif

#include <stdio.h>
#include <xom.h>
#include <xds.h>
#include <xdsbdcp.h>
#include <xdsgds.h>
#include <xdscds.h>
#include <xdsext.h>         /* convenience functions header file  */
#include <xomext.h>         /* convenience functions header file  */
#include "acl2.h"
void
main(
    int  argc,
    char *argv[]
)
{
  OM_workspace       workspace;     /* Workspace for objects          */
  OM_private_object  session;       /* Session object.                */
  OM_private_object  bound_session; /* Holds the Session object which */
                                    /* is returned by ds_bind()       */
  OM_private_object  context;       /* Context object.                */
  OM_private_object  result;        /* Holds the search result object.*/
  OM_sint            invoke_id;     /* Integer for the invoke id      */
                                    /* returned by ds_search().       */
                                    /* (this parameter must be present*/
                                    /* even though it is ignored).    */
  OM_type            navigation_path[] = { DS_SEARCH_INFO, 0 };
                                    /* List of OM types to the target */
                           /* object - of the search result  */
    OM_type            entry_list[] = { DS_ENTRIES, 0 };
                                    /* List of types to be extracted  */
  OM_public_object   entry;         /* Entry object from search info. */
  OM_value_position  total_num;     /* Number of descriptors returned.*/
  OM_return_code     rc;            /* XOM function return code.      */
```

```
register int       i;
char               user_name[MAX_DN_LEN] = DN_NORBERT;
                              /* Holds the requestor's name -   */
                              /* "/C=de/O=sni/OU=ap/CN=norbert" */
char               entry_string[MAX_DN_LEN + 7] = "[?r??] ";
                              /* Holds entry details.           */
struct entry       entry_array[6];/* List of entry names and attrs  */
OM_object        credentials; /* Credentials part of session obj*/
OM_object        use_copy;    /* Specifies whether to use shadow*/
         /* entries, in context object     */
OM_object        filter;      /* Filter - for search operation  */
OM_object        dn_root;     /* Name object for "/"       */
OM_object        selection_acl;   /* Entry Information          */
      /* Selection obj              */

static char        *name_list[] =
{ DN_DE, DN_SNI, DN_AP, DN_STEFANIE,
                             DN_NORBERT, DN_INGRID };
                            /* Array of names to be added   */
static char        *C_attr_list[]  = { OBJ_CLASS_C };
static char        *O_attr_list[]  = { OBJ_CLASS_O, ATT_ACL1 };
static char        *OU_attr_list[] = { OBJ_CLASS_OU };
static char        *OP_attr_list[] = { OBJ_CLASS_OP, ATT_ACL2,
                         ATT_SURNAME, ATT_PHONE_NUM, ATT_PASSWORD };
                              /* Attribute lists, in string fmt */

static char        *dn_root_str    = DN_ROOT;
static char        *filter_str     = FILTER;

/* Step 3   *
 * Initialize a directory workspace for use by XOM.   */
if ((workspace = ds_initialize()) == (OM_workspace)0)
    printf("ds_initialize() error\n");

/* Step 4   *
 * Negotiate the use of the BDC and GDS packages.   */
if (ds_version(features, workspace) != DS_SUCCESS)
    printf("ds_version() error\n");

/* Step 5   *
 * Build name objects for entries to be added to the directory.   */
for (i = 0; i < NO_OF_ENTRIES; i++)
    if (! build_name_object(workspace,name_list[i],
                            &(entry_array[i].name)))
    printf("build_name_object() error\n");

/* Step 6   *
 * Build attribute objects for entries to be added to the directory   */
if ((! build_attr_list_object(workspace, NO_C_ATTRS, C_attr_list,
                              &entry_array[0].attr_list)) ||
    (! build_attr_list_object(workspace, NO_O_ATTRS, O_attr_list,
                              &entry_array[1].attr_list)) ||
    (! build_attr_list_object(workspace, NO_OU_ATTRS, OU_attr_list,
                              &entry_array[2].attr_list)) ||
    (! build_attr_list_object(workspace, NO_OP_ATTRS, OP_attr_list,
                              &entry_array[3].attr_list)))
    printf("build_attr_list_object() error\n");
/*
 * These entries also have the OP attribute list.
 */
entry_array[4].attr_list = entry_array[3].attr_list;
entry_array[5].attr_list = entry_array[3].attr_list;

/* Step 7
 *
 * Add a fixed tree of entries to the directory in order to permit
 * an authenticated bind by:   /C=de/O=sni/OU=ap/CN=norbert
```

```
     */
    if (! add_tree(workspace, entry_array, NO_OF_ENTRIES))
        printf("add_tree() error\n");

    /* Step 8
     *
     * Create a default session object.
     */
    if ((rc = om_create(DSX_C_GDS_SESSION,OM_TRUE,workspace,&session))
                                                  != OM_SUCCESS)
        printf("om_create() error %d\n", rc);

    /*  Step 9
     *
     * Build an object with the following credentials:
     *  requestor:  /C=de/O=sni/OU=ap/CN=norbert
     *  password:   "secret"
     *  authentication mechanism: simple
     */
    if (! build_credentials_object(entry_array[4].name,&credentials))
        printf("build_credentials_object() error\n");

    /*
     * Alter the default session object to include the credentials
     */
    if ((rc = om_put(session, OM_REPLACE_ALL, credentials, 0 ,0, 0))
                                                != OM_SUCCESS)
        printf("om_put() error %d\n", rc);

    /* Step 10
     *
     * Bind with credentials to the default GDS server.  The
     * returned session object is stored in the private object variable
     * bound_session and is used for all further XDS function calls.
     */
    if (ds_bind(session, workspace, &bound_session) != DS_SUCCESS)
        printf("ds_bind() error\n");

    /* Step 11
     *
     * Create a default context object.
     */
    if ((rc = om_create(DSX_C_GDS_CONTEXT,OM_TRUE,workspace,&context))
                                                  != OM_SUCCESS)
        printf("om_create() error %d\n", rc);

    /*
     * Build an object specifying that shadow entries should be used.
     */
    if (! build_use_copy_object(&use_copy))
        printf("build_use_copy_object() error\n");

    /*
     * Alter the default context object to include 'shadow' entries.
     */
    if ((rc = om_put(context, OM_REPLACE_ALL, use_copy, 0 ,0, 0))
                   != OM_SUCCESS) printf("om_put() error %d\n",
  rc);

    /* Step 12
     *
     * Build a filter object, specifying presence of object class attr.
     */
    if (! build_filter_object(workspace, filter_str, &filter))
        printf("build_filter_object() error\n");

    /*
```

```
 * Build a root name object, name = "/"
 */
if (! build_name_object(workspace, dn_root_str, &dn_root))
    printf("build_name_object() error\n");

/*
 * Build an entry information selection object,
 * selecting acl attributes.
 */
if (! build_selection_object(&selection_acl))
    printf("build_selection_object() error\n");
/* Step 13
 *
 * Search the whole subtree below root.  The filter selects entries
 * with an object-class attribute.  The selection extracts the ACL
 * attribute from each selected entry.  The results are returned in
 * the private object 'result'.
 *
 * NOTE: Since every entry contains an object-class attribute the
 *       filter performs no function other than to demonstrate how
 *       filters may be used.
 */

if (ds_search(bound_session, context, dn_root, DS_WHOLE_SUBTREE,filter,
        OM_FALSE, selection_acl, &result, &invoke_id) !=DS_SUCCESS)
    printf("ds_search() error\n");

/* Step 14
 *
 * Close the connection to the GDS server.
 */
if (ds_unbind(bound_session) != DS_SUCCESS)
    printf("ds_unbind() error\n");

/* Step 15
 *
 * Remove the user's credentials from the directory.
 */
if (! remove_tree(workspace, session, entry_array, NO_OF_ENTRIES))
    printf("remove_tree() error\n");

/* Step 16
 *
 * Free the name and attribute objects
 * which make up the directory entries.
 */
if (! free_entry_list(entry_array))
    printf("free_entry_list() error\n");
/*
 * Free public objects which were created.
 */
free(selection_acl);
free(use_copy);
free(credentials);

if ((om_delete(filter) != OM_SUCCESS) ||
    (om_delete(dn_root) != OM_SUCCESS))
    printf("om_delete() error\n");

/* Step 17
 *
 * Extract components from the search result by means of the XOM
 * Convenience Function, omX_extract()
 */
if ((rc = omX_extract(result, navigation_path,
            OM_EXCLUDE_ALL_BUT_THESE_TYPES + OM_EXCLUDE_SUBOBJECTS,
            entry_list, OM_FALSE, 0, 0, &entry, &total_num))
```

```
                                                  != OM_SUCCESS)
        printf("omX_extract(Search-Result) error %d\n", rc);

    /*
     * Requestor's name = "/C=de/O=sni/OU=ap/CN=norbert"
     */
    printf("User:  %s\nTotal: %d\n", user_name, total_num);

    /* Step 18
     *
     * Examine each entry and print the entry details.
     */
    for (i = 0; i < total_num; i++) {
        if (process_entry_info((entry+i)->value.object.object,
                           entry_string, user_name))
            printf("%s\n", entry_string);
    }

    /*
     * Now free the entry object (returned from omX_extract() ).
     */
    if (om_delete(entry) != OM_SUCCESS)
        printf("om_delete() error\n");
    /* Step 19
     *
     * Close the directory workspace.
     */
    if (ds_shutdown(workspace) != DS_SUCCESS)
        printf("ds_shutdown() error\n");
  }

  /*
   * Add the tree of entries described above.
   */
  int
  add_tree(
      OM_workspace workspace,
      struct entry elist[],
      int          no_entries
  )
  {
    OM_private_object  session;       /* Holds the Session object which */
                                      /* is returned by ds_bind()       */
    OM_sint            invoke_id;     /* Integer for the invoke id      */
    int                error = 0;
    int                i;

    /*
     * Bind (without credentials) to the default GDS server.
     */
    if (ds_bind(DS_DEFAULT_SESSION, workspace, &session) !=
  DS_SUCCESS)
        error++;

    /*
     * Add entries to the GDS server.
     */
    for (i = 0; i < no_entries; i++)
        if (ds_add_entry(session, DS_DEFAULT_CONTEXT, elist[i].name,
            elist[i].attr_list, &invoke_id) != DS_SUCCESS) {
        /* Ignore error if adding country - possibly already there      */
            if (i != 0) error++;
        }
    /*
     * Close the connection to the GDS server.
     */
    if (ds_unbind(session) != DS_SUCCESS)
```

```
      error++;

  return (error?0:1);
}

/*
 * Remove the tree of entries described above.
 */
int
remove_tree(
    OM_workspace       workspace,
    OM_private_object  session,
    struct entry       elist[],
    int                no_entries
)
{
  OM_private_object bound_session;  /* Holds the Session object which */
                                    /* is returned by ds_bind()       */
  OM_sint           invoke_id;      /* Integer for the invoke id      */
  int               i;
  int               error = 0;

  /*
   * Bind (without credentials) to the default GDS server.
   */
  if (ds_bind(session, workspace, &bound_session) != DS_SUCCESS)
      error++;

  /*
   * Remove entries from the GDS server.
   */
  for (i = no_entries-1; i >= 0; i--)
   if (ds_remove_entry(bound_session, DS_DEFAULT_CONTEXT,
              elist[i].name, &invoke_id) != DS_SUCCESS) {
         /* Ignore error if removing country - possibly has entries  */
         /* below it      */
         if (i != 0) error++;
      }
  /*
   * Close the connection to the GDS server.
   */
  if (ds_unbind(bound_session) != DS_SUCCESS)
      error++;

  return (error?0:1);
}

/*
 * Extract information about an entry from the Entry-Info object:
whether
 * the entry is a master-copy, its ACL permissions and its distinguished

 * name.  Build up a string based on this information.
 */
int
process_entry_info(
  OM_private_object  entry,
  char              *entry_string,
  char              *user_name
)
{
  OM_return_code     rc;            /* Return code from XOM function. */
  OM_public_object   ei_attrs;      /* Components from Entry-Info.    */
  OM_public_object   attr;          /* Directory attribute.           */
  OM_public_object   acl;           /* ACL attribute parts.           */
  OM_public_object   acl_vals;      /* ACL attribute value.           */
  OM_public_object   acl_item;      /* ACL item component.            */
  OM_value_position  total_attrs;   /* Number of attributes returned. */
  OM_value_position  total_acls;    /* Number of acl values returned. */
  register int       i;
  register int       interp;
  register int       error = 0;
  register int       found_acl = 0;
  static OM_type     ei_attr_list[] = { DS_FROM_ENTRY,
                                        DS_OBJECT_NAME,
                                        0 };
                                    /* Attributes to be extracted.    */
  OM_string        entry_str;
  /*
   * Extract occurrences of DS_FROM_ENTRY, and DS_OBJECT_NAME
   * from each Entry-Info object.
   */
  if ((rc = om_get(entry, OM_EXCLUDE_ALL_BUT_THESE_TYPES,
               ei_attr_list, OM_FALSE, 0, 0, &ei_attrs,
&total_attrs))
                                                       != OM_SUCCESS) {
      error++;
      printf("om_get(Entry-Info) error %d\n", rc);
  }

  for (i = 0; ((i < total_attrs) && (! error)); i++,
ei_attrs++) {

      /*
       * Determine if current entry is a master-copy or a shadow-copy.
       */
      if ((ei_attrs->type == DS_FROM_ENTRY) &&
          ((ei_attrs->syntax & OM_S_SYNTAX) == OM_S_BOOLEAN))
          if (ei_attrs->value.boolean == OM_TRUE)
              entry_string[1] = 'm';
          else if (ei_attrs->value.boolean == OM_FALSE)
                   entry_string[1] = 's';
              else
                   entry_string[1] = '?';

      /*
       * Convert the entry's distinguished name to a string format.
       */
      entry_str.elements = &entry_string[7];
      entry_str.length = MAX_DN_LEN;
      if ((ei_attrs->type == DS_OBJECT_NAME) &&
          ((ei_attrs->syntax & OM_S_SYNTAX) == OM_S_OBJECT))
          if ((rc = omX_object_to_string(ei_attrs->value.object.object,
                    OM_FALSE, &entry_str)) != OM_SUCCESS) {
              error++;
              printf("omX_object_to_string() error\n");
          }
  }
  /*
   * Now extract occurences of attributes, where the attribute
   * type is ACL from the Entry-Info object.
   */
  dsX_extract_attr_values(entry, DSX_A_ACL, OM_TRUE,
    &acl_vals, &total_acls);

  for (i = 0; ((i < total_acls) && (! error)); i++) {
      acl = acl_vals[i].value.object.object;

      /*
       * Examine the ACL. Check each permission for the current user.
       */

      entry_string[2] = 'r';
      entry_string[3] = '-';
```

```
        entry_string[4] = '-';

        while (acl->type != OM_NO_MORE_TYPES) {

            if ((acl->syntax & OM_S_SYNTAX) == OM_S_OBJECT)
                acl_item = acl->value.object.object;

            switch (acl->type) {

                case OM_CLASS:
                    break;

                case DSX_MODIFY_PUBLIC:
                    if (permitted_access(user_name, acl_item))
                        entry_string[2] = 'w';
                    break;

                case DSX_READ_STANDARD:
                    if (permitted_access(user_name, acl_item))
                        entry_string[3] = 'r';
                    break;

                case DSX_MODIFY_STANDARD:
                    if (permitted_access(user_name, acl_item))
                        entry_string[3] = 'w';
                    break;
                case DSX_READ_SENSITIVE:
                    if (permitted_access(user_name, acl_item))
                        entry_string[4] = 'r';
                    break;

                case DSX_MODIFY_SENSITIVE:
                    if (permitted_access(user_name, acl_item))
                        entry_string[4] = 'w';
                    break;
            }
            acl++;
        }
    }
    /*
     * Now free acl_vals.
     */
    if (total_acls > 0)
     if ((rc = om_delete(acl_vals)) != OM_SUCCESS) {
            error++;
            printf("om_delete() error, rc = %d\n", rc);
     }

    return (error?0:1);
}

/*
 * Check if a user is permitted access based on the ACL supplied.
 */
int
permitted_access(
    char             *user_name,
    OM_public_object  acl_item
)
{
    char                 acl_name[MAX_DN_LEN];
    OM_string            acl_name_str;
    int                  interpretation;
    int                  acl_present = 0;
    int                  access = 0;
    int                  acl_name_length;
    OM_return_code       rc;
```

```
  while (acl_item->type != OM_NO_MORE_TYPES) {

      switch (acl_item->type) {
      case OM_CLASS:
          break;

      case DSX_INTERPRETATION:
          interpretation = acl_item->value.boolean;
          break;

      case DSX_USER:
          acl_name_str.elements = acl_name;
          if ((rc = omX_object_to_string(acl_item->value.object.object,
                  OM_FALSE, &acl_name_str)) == OM_SUCCESS) {
                if (interpretation == DSX_SINGLE_OBJECT) {
                if (strcmp(acl_name, user_name) == 0)
                      access = 1;
           }
           else if (interpretation == DSX_ROOT_OF_SUBTREE) {
                     if ((acl_name_length = strlen(acl_name)) == 0)
                         access = 1;
                     else if
(strncmp(acl_name,user_name,acl_name_length)
  == 0)
                         access = 1;
              }
  }
          break;
      }
      acl_item++;
  }

  return (access);
}

/*
 * Build a name object from a name string using the XOM
 * Convenience Function omX_string_to_object().
 */
int
build_name_object(
  OM_workspace  workspace,
  char         *name,
  OM_private_object   *name_obj
)
{
  OM_integer        err_pos;
  OM_integer        err_type;
  OM_return_code    rc;
  OM_string         name_str;
  int               error = 0;
  name_str.length = strlen(name);
  name_str.elements = name;
  if ((rc = omX_string_to_object(workspace, &name_str, DS_C_DS_DN,
                 OM_TRUE, name_obj, &err_pos, &err_type)) !=
OM_SUCCESS)
      error++;

  return (error?0:1);
}


/*
 * Build an attribute list object given a list of attribute strings.
 * Use the XOM Convenience Function omX_string_to_object() to build
 * an attribute object from an attribute string, and omX_fill() to
 * create the other OM descriptor required.
 */
int
build_attr_list_object(
  OM_workspace  workspace,
  OM_integer   no_attrs,
  char   *attr_str_array[],
  OM_object   *attr_list_obj
)
{
  OM_integer        err_pos;
  OM_integer        err_type;
  OM_object       attr;
  OM_object       alist;
  OM_string       attr_str;
  OM_return_code    rc;
  OM_descriptor     null_desc = OM_NULL_DESCRIPTOR;
  int               error = 0;
  int               i;

  /*
   * Allocate space for class descriptor, null descriptor and
   * one descriptor for each attribute.
   */
  if ((alist =
    (OM_descriptor *)malloc((2+no_attrs) * sizeof(OM_descriptor)))
                        == 0)
      error++;

  if ((rc = omX_fill_oid(OM_CLASS, DS_C_ATTRIBUTE_LIST, &alist[0]))
                       != OM_SUCCESS)
      error++;
  for (i = 1; i <= no_attrs; i++) {

      attr_str.length = strlen(attr_str_array[i-1]);
      attr_str.elements = attr_str_array[i-1];
      if ((rc = omX_string_to_object(workspace, &attr_str,
DS_C_ATTRIBUTE,
           OM_TRUE, &attr, &err_pos, &err_type)) !=
OM_SUCCESS)
      error++;

      if ((rc = omX_fill(DS_ATTRIBUTES, OM_S_OBJECT, 0, attr,
&alist[i]))
           != OM_SUCCESS)
      error++;
  }

  alist[i] = null_desc;

  *attr_list_obj = alist;
  return (error?0:1);
}


/*
 * Build an entry info selection object using the XOM Convenience
 * Functions omX_fill() and omX_fill_oid() to fill the OM descriptors.
 */
int
build_selection_object(
  OM_object  *selection_obj
)
{
  OM_integer        err_pos;
  OM_integer        err_type;
  OM_object         desc;
  OM_object         sel;
  OM_return_code    rc;
  OM_descriptor     null_desc = OM_NULL_DESCRIPTOR;
  int               error = 0;

  /*
   * Allocate space for class descriptor, null descriptor and one
   * descriptor for each attribute.
   */
  if ((sel = (OM_descriptor *)malloc((5) * sizeof(OM_descriptor))) == 0)
      error++;
  if ((rc = omX_fill_oid(OM_CLASS, DS_C_ENTRY_INFO_SELECTION,
&sel[0]))
                    != OM_SUCCESS)
      error++;

  if ((rc = omX_fill(DS_ALL_ATTRIBUTES, OM_S_BOOLEAN, OM_FALSE, 0,
                                        &sel[1])) != OM_SUCCESS)
      error++;

  if ((rc = omX_fill_oid(DS_ATTRIBUTES_SELECTED, DSX_A_ACL,
                                        &sel[2])) != OM_SUCCESS)
      error++;

  if ((rc = omX_fill(DS_INFO_TYPE, OM_S_ENUMERATION,
DS_TYPES_AND_VALUES,
     0, &sel[3])) != OM_SUCCESS)
    error++;

  sel[4] = null_desc;

  *selection_obj = sel;
  return (error?0:1);
}

/*
 * Build a credentials object using the XOM Convenience Function
 * omX_fill().
 */
int
build_credentials_object(
  OM_object  name,
  OM_object     *credentials_obj
)
{
  OM_integer        err_pos;
  OM_integer        err_type;
  OM_object         cred;
  OM_return_code    rc;
  OM_descriptor     null_desc = OM_NULL_DESCRIPTOR;
  int               error = 0;

  /*
   * Just allocate space for a null descriptor and two other
descriptors,
   * no class descriptor required.
   */
  if ((cred = (OM_descriptor *)malloc((4) * sizeof(OM_descriptor))) ==
0)
      error++;
  if ((rc = omX_fill(DS_REQUESTOR, OM_S_OBJECT, 0, name, &cred[0]))
                                                     != OM_SUCCESS)
      error++;

  if ((rc = omX_fill(DSX_PASSWORD, OM_S_OCTET_STRING,
(sizeof(PASSWD)-1),
                                  PASSWD, &cred[1])) != OM_SUCCESS)
  if ((rc = omX_fill(DSX_AUTH_MECHANISM, OM_S_ENUMERATION, DSX_SIMPLE,
                                     0, &cred[2])) != OM_SUCCESS)
      error++;

  cred[3] = null_desc;

  *credentials_obj = cred;
  return (error?0:1);
}

/*
 * Build an object setting DS_DONT_USE_COPY to FALSE, using the
 * XOM Convenience Function omX_fill().
 */
int
build_use_copy_object(
  OM_object        *use_copy_obj
)
{
  OM_integer        err_pos;
  OM_integer        err_type;
  OM_object         desc;
  OM_object         copy;
  OM_return_code    rc;
  OM_descriptor     null_desc = OM_NULL_DESCRIPTOR;
  int               error = 0;

  /*
   * Just allocate space for a null descriptor and one other
   * descriptor, no class descriptor required.
   */
  if ((copy = (OM_descriptor *)malloc((2) * sizeof(OM_descriptor))) ==
0)
  error++;

  if ((rc = omX_fill(DS_DONT_USE_COPY, OM_S_BOOLEAN, OM_FALSE, 0,
                   &copy[0])) != OM_SUCCESS)
      error++;

  copy[1] = null_desc;
  *use_copy_obj = copy;
  return (error?0:1);
}


/*
 * Build a filter object from a filter string using the XOM Convenience
 * Function omX_string_to_object().
 */
int
build_filter_object(
  OM_workspace workspace,
  char   *filter,
  OM_object  *filter_obj
)
{
  OM_integer        err_pos;
  OM_integer        err_type;
  OM_string         filter_str;
  OM_return_code    rc;
  int               error = 0;

  filter_str.length = strlen(filter);
  filter_str.elements = filter;
  if ((rc = omX_string_to_object(workspace, &filter_str,
DS_C_FILTER,
              OM_TRUE, filter_obj, &err_pos, &err_type)) !=
OM_SUCCESS)
      error++;

  return (error?0:1);
}


/*
 * Free the name and attribute list objects in the entry list.  Objects
 * which have been created using the XOM Convenience Function
 * omX_string_to_object() must be deleted using om_delete().
 */
int
free_entry_list(
  struct entry       entry_array[]
)
{
  OM_object       attr_list_obj;
  int                 i, j;
  int                 error = 0;
  for (i = 0; i < NO_OF_ENTRIES; i++)  {

      /*
       * Delete the service generated public name object .
       */
      if (om_delete(entry_array[i].name) != OM_SUCCESS)
          error++;

      /*
       * The last two attribute lists were the same as the 4th one.
       */
      if (i < NO_OF_ENTRIES-2) {
          attr_list_obj = entry_array[i].attr_list;
          for (j = 0; attr_list_obj[j].type != OM_NO_MORE_TYPES; j++) {
              if (attr_list_obj[j].type == DS_ATTRIBUTES)
           /*
            * Delete the service generated public attribute object.
            */
           if (om_delete(attr_list_obj[j].value.object.object)
                != OM_SUCCESS)
               error++;
          }

          /*
           * Free the whole attribute list object.
           */
            free(attr_list_obj);
      }
  }

  return (error?0:1);
}
```

The acl2.h Header File

The **acl2.h** header file performs the following:

1. It exports the object identifiers that **acl2.c** requires.
2. It declares a structure to contain the name and attributes of directory entries.
3. It defines abbreviated names for the directory entries.
4. It defines abbreviated names for the directory attributes.
5. It builds the descriptor list for optional packages that are to be negotiated.

The following code is a listing of the **acl2.h** file:

```
/****************************************************************
*                                                               *
*  COPYRIGHT (C) SIEMENS NIXDORF INFORMATIONSSYSTEME AG 1991   *
*               ALL RIGHTS RESERVED                             *
*                                                               *
***************************************************************/

#ifndef _ACL2_H
#define _ACL2_H

#define MAX_DN_LEN 100  /* max length of a distinguished name in  */
                        /* string format.                         */

/* Step 1 */

/* The application must export the object identifiers it requires. */

OM_EXPORT (DS_C_DS_DN)
OM_EXPORT (DS_C_ENTRY_INFO_SELECTION)
OM_EXPORT (DS_C_ATTRIBUTE)
OM_EXPORT (DS_C_ATTRIBUTE_LIST)
OM_EXPORT (DS_C_FILTER)
OM_EXPORT (DSX_C_GDS_SESSION)
OM_EXPORT (DSX_C_GDS_CONTEXT)
OM_EXPORT (DSX_A_ACL)

/* Structure to contain the name and attribute list               */
/* of a directory entry.                                          */

struct entry {
    OM_private_object   name;
    OM_object           attr_list;
} Entry;
/* Step 2  */
/*
 * Names of directory entries, in string format.
 */
#define DN_ROOT         "/"
#define DN_DE           "/C=de"
#define DN_SNI          "/C=de/O=sni"
#define DN_AP           "/C=de/O=sni/OU=ap"
#define DN_STEFANIE     "/C=de/O=sni/OU=ap/CN=stefanie"
#define DN_NORBERT      "/C=de/O=sni/OU=ap/CN=norbert"
#define DN_INGRID       "/C=de/O=sni/OU=ap/CN=ingrid"

/*
 * Attributes, in string format.
 */
#define OBJ_CLASS_C     "OCL = TOP; C"
#define OBJ_CLASS_O     "OCL = TOP; ORG"
#define OBJ_CLASS_OU    "OCL = TOP; OU"
#define OBJ_CLASS_OP    "OCL = TOP; PER; ORP"
#define ATT_PHONE_NUM   "TN = '+49 89 636 0' "
#define ATT_PASSWORD    "UP = secret"
#define ATT_SURNAME     "SN = Schmid"
#define ATT_ACL1  "ACL={MPUB = {INT = 1,USR = {/}}, \
          RSTD = {INT = 0,USR = {/C=de/O=sni/OU=ap/CN=stefanie}},\
          MSTD = {INT = 0,USR = {/C=de/O=sni/OU=ap/CN=stefanie}},\
          RSEN = {INT = 0,USR = {/C=de/O=sni/OU=ap/CN=stefanie}},\
          MSEN = {INT = 0,USR = {/C=de/O=sni/OU=ap/CN=stefanie}}}"
#define ATT_ACL2  "ACL={MPUB = {INT = 1,USR =
{/C=de/O=sni/OU=ap}},\
          RSTD = {INT = 1,USR = {/C=de/O=sni/OU=ap}},\
          MSTD = {INT = 0,USR = {/C=de/O=sni/OU=ap/CN=stefanie}},\
          RSEN = {INT = 1,USR = {/C=de/O=sni/OU=ap}},\
          MSEN = {INT = 0,USR = {/C=de/O=sni/OU=ap/CN=stefanie}}}"
```

```
/* Other strings.                                              */
#define PASSWD          "secret"
#define FILTER          "OCL"

#define NO_OF_ENTRIES   6   /* 6 entries to be added             */
#define NO_C_ATTRS      1   /* 1 attr in Country attribute list   */
#define NO_O_ATTRS      2   /* 2 attr in Org attribute list       */
#define NO_OU_ATTRS     1   /* 1 attr in Org-Unit attribute list  */
#define NO_OP_ATTRS     5   /* 5 attr in Org-Person attribute list*/
/* Build up an array of object identifiers for the optional      */
/* packages to be negotiated.                                    */
DS_feature features[] = {
    { OM_STRING(OMP_O_DS_BASIC_DIR_CONTENTS_PKG), OM_TRUE },
    { OM_STRING(OMP_O_DSX_GDS_PKG), OM_TRUE },
    { 0 }
};

#endif /* _ACL2_H
*/
```

Example Strings

This section contains examples of input strings to **omX_string_to_object( )** and some examples of strings that can be returned by **omX_object_to_string( )**.

Input Strings to omX_string_to_object( )

The following are examples of strings that can be handled by the **omX_string_to_object( )** function.

Example 1: To create a `DS_C_DS_DN` object (root), use strings like the following:

```
/
/...
```

Example 2: To create other `DS_C_DS_DN` objects, use strings like the following:

```
/c=de/o=sni/ou=ap11/cn=naik,sn=naik
/c=de/o=sni/ou=ap11/85.4.3=naik,sn=naik
/c=de/o=sni/ou=ap11/cn=naik,sn=na\x69k
/c=de/o=sni/ou=ap11/cn=naik,loc=Muenchen\,8000
/c=de/o=sni/ou=ap11/cn=naik,loc='Muenchen,8000'
/ C = de / O = sni / Ou = ap11/CN=naik,
SN=naik
```

Example 3: To create a `DS_C_DS_DN` object (DCE name), use a string like the following:

```
/.../c=us/o=osf/ou=abc/subsystems/server/xyz
```

Example 4: To create a `DS_C_DS_RDN` object, use strings like the following:

```
cn=naik,sn=naik
cn=naik,sn=na\x69k
CN = naik, SN = naik
```

Example 5: To create a `DS_C_DS_RDN` object (DCE name), use a string like the following:

```
server
```

Example 6: To create a `DS_C_ATTRIBUTE` object (containing, for example, `Common-Name`), use strings like the following:

```
cn=bhavesh naik
CN = bhavesh naik
85.4.3=bhavesh nai\x69k
```

Example 7: To create a `DS_C_ATTRIBUTE` object (containing an object class with multiple values of `Residential-Person` and `Organizational-Person`), use strings like the following:

```
OCL=REP;ORP
OCL = '\x55\x06\x0a' ;
'\x55\x06\x07'
```

Example 8: To create a `DS_C_ATTRIBUTE` object (containing a GDS structured attribute like `Telex-Number` or `Owner`), use strings like the following:

```
TXN={TN=12345,CC=678,AB=90}
TXN = { TN = 12345, CC = 678, AB = 90}
own={/c=de/o=sni/ou=ap11};{/c=de/o=sni/ou=ap22}
pa={pa='Wilhelm Riehl Str.85';'Munich'}
```

Example 9: To create a `DSX_C_GDS_ACL` object, use a string like the following:

```
MPUB={INT=0, USR={/c=de/o=sni/cn=naik,
sn=bhavesh}}
```

Example 10: To create a `DS_C_PRESENTATION_ADDRESS` object, use a string like the following:

```
TS=Server,NA='TCP/IP!internet=127.0.0.1+port=25015'
```

Example 11: To create a `DS_C_FILTER` object, use strings like the following:

```
c
!c
C = de && CN = 'bha\x76esh naik'
c=de&&cn ~ =mueller
c = de && (cn = 'a*' || cn = b* || cn = c* )
ACL={MPUB={INT=0,USR={/c=de/o=sni/cn=naik, sn=bhavesh}}}
c = de || cn = *aa*bb*cc*
(cn ~ =naik)&&((OCL=ORP)||(OCL=REP))&&
!(SN='bhavesh naik')&&(L=*)
```

Example 12: The following is an example of the error return when an erroneous string is supplied:

```
/c=de/o=sni,=de
```

The `OM_return_code` would be `OM_WRONG_VALUE_MAKEUP`.

The `error_type` would be `OMX_MISSING_ABBRV`.

The `error_position` would be 13.

Strings Returned by omX_object_to_string( )

The following are examples of strings returned by the **omX_object_to_string( )** function.

Example 1: If a DS_C_DS_DN object is supplied, the following might be returned:

```
/
/C=de/O=sni/OU=ap11/CN=naik,SN=naik
/C=de/O=sni/OU=ap11/CN=naik,LOC=Muenchen\,8000
```

Example 2: If a DS_C_DS_RDN object is supplied, the following might be returned:

```
CN=naik,SN=naik
server
```

Example 3: If a DS_C_ATTRIBUTE object is supplied, the following might be returned:

```
CN=bhavesh naik
OCL=REP;ORP
TXN={AB=90,CC=678,TN=12345}
OWN={/C=de/O=sni/OU=ap11};{/C=de/O=sni/OU=ap22}
```

Example 4: If a DSX_C_GDS_ACL object is supplied, the following might be returned:

```
MPUB={INT=0,USR={/C=de/O=sni/CN=naik,SN=bhavesh}}
```

Example 5: If a DS_C_NAME_ERROR object is supplied with DS_PROBLEM of DS_E_NO_SUCH_OBJECT, the following might be returned:

```
The specified name does not match the name of any object
in the directory
```

Example 6: If a DS_C_ATTRIBUTE_ERROR object is supplied with DS_C_ATTRIBUTE_PROBLEM containing DS_E_ATTRIBUTE_OR_VALUE_EXISTS, the following might be returned:

```
An attempt is made to add an attribute or value that already
exists.  Violating Attribute -
Telephone-Number
```

Part 4. XDS/XOM Supplementary Information

This section contains reference material for the X/Open Object Management (XOM) programming interface.

Chapter 9. XDS Interface Description

The XDS interface comprises a number of functions, together with many OM classes of OM objects, which are used as the parameters and results of the functions. Both the functions and the OM objects are based closely on the abstract service that is specified in the standards (see *The Directory: Abstract Service Definition*, ISO 9594-3, CCITT X.511).

The interface models the directory interactions as service requests made through a number of interface functions, which take a number of input parameters. Each valid request causes an operation within the directory service, which eventually returns a status and any result of the operation.

All interactions between the user and the directory service belong to a session, which is represented by an OM object passed as the first parameter to most interface functions.

The other parameters to the functions include a context and various service-specific parameters. The context includes a number of parameters that are common to many functions, and that seldom change from operation to operation.

Each of the components of this model are described in the following sections in this chapter along with other features of the interface, such as security.

XDS Conformance to Standards

The XDS interface defines an API that application programs can use to access the functionality of the underlying directory service. The DCE XDS API conforms to the *X/Open CAE Specification, API to directory services (XDS)* (November 1991).

The DCE XDS implementation supports the following features:

- A synchronous interface. Asynchronous functionality can be achieved by using threads as described in “Chapter 7. Using Threads With The XDS/XOM API” on page 145.
- All synchronous interface functions except **ds_search** and **ds_modify_rdn** are supported. The two asynchronous-specific functions are handled as follows: (for all practical purposes, these functions are not supported)
  - **ds_abandon( )**

    This call does not issue a directory service abandon operation. It returns with a DS_C_ABANDON_FAILED (DS_E_TOO_LATE) error.
  - **ds_receive_result( )**

    If there are any outstanding operations (when multiple threads issue XDS calls in parallel), this function returns DS_SUCCESS with the *completion_flag_return* parameter set to DS_OUTSTANDING_OPERATIONS. If no XDS calls are outstanding, this function returns DS_SUCCESS with the *completion_flag_return* parameter set to DS_NO_OUTSTANDING_OPERATION.
- Automatic connection management is not provided. The **ds_bind( )** and **ds_unbind( )** functions always try, respectively, to set up and release directory service connections immediately.
- The DS_FILE_DESCRIPTOR attribute of the DS_C_SESSION object is not used.
- The default values for OM attributes in the DS_C_CONTEXT and DS_C_SESSION objects are described in “Chapter 10. XDS Class Definitions” on page 189.

DCE XDS supports four packages, where one is mandatory and three are optional. Use of the optional packages is negotiated by using **ds_version( )**. The packages are as follows:

- The Directory Service Package (as defined in "Chapter 10. XDS Class Definitions" on page 189), which also includes the errors. This package is required.
- The Basic Directory Contents Package (as defined in "Chapter 11. Basic Directory Contents Package" on page 219). This package is optional and is not required for XDS/XOM over CDS. However, the **xdsbdcp.h** header file is required if any X.500 cell names are used.
- The Global Directory Service Package is optional and is not required for XDS/XOM over CDS.
- The MHS Directory User Package is optional and is not required for XDS/XOM over CDS.

None of the OM classes defined in these four packages are encodable. Thus, DCE XDS application programmers do not require the use of the XOM functions **om_encode( )** and **om_decode( )**, which are not supported by the DCE XOM API.

The XDS Functions

As mentioned already, the standards define abstract services that requestors use to interact with the directory. Each of these abstract services maps to a single function call, and the detailed specifications are given in the XDS reference pages. The services and the function calls to which they map are as follows:

`DirectoryBind`
: Maps to **ds_bind( )**

`DirectoryUnbind`
: Maps to **ds_unbind( )**

`Read`
: Maps to **ds_read( )**

`Compare`
: Maps to **ds_compare( )**

`Abandon`
: Maps to **ds_abandon( )** (not supported)

`List`
: Maps to **ds_list( )**

`Search`
: Maps to **ds_search( )** (not supported for XDS over CDS)

`AddEntry`
: Maps to **ds_add_entry( )**

`RemoveEntry`
: Maps to **ds_remove_entry( )**

`ModifyEntry`
: Maps to **ds_modify_entry( )**

`ModifyRDN`
: Maps to **ds_modify_rdn( )** (not supported for XDS over CDS)

There is a function called **ds_receive_result( )**, which has no counterpart in the abstract service. It is used with asynchronous operations. (See the **xds_intro(3xds)** reference page for information on how the asynchronous functions **ds_abandon( )** and **ds_receive_result( )** are handled by the DCE XDS API.)

The **ds_initialize( )**, **ds_shutdown( )**, and **ds_version( )** functions are used to control the XDS API and do not initiate any directory operations.

The interface functions are summarized in Table 16.

Table 16. The XDS Interface Functions

Name	Description
ds_abandon( )	Abandons the result of a pending asynchronous operation. This function is not supported. See **xds_intro(3xds)**.
ds_add_entry( )	Adds a leaf entry to the CDS namespace.
ds_bind( )	Opens a session with CDS namespace.
ds_compare( )	Compares a purported attribute value with the attribute value stored in CDS object.
ds_initialize( )	Initializes the XDS interface.
ds_list( )	Enumerates the names of the immediate subordinates of a particular directory entry.
ds_modify_entry( )	Atomically performs modification to a directory entry.
ds_modify_rdn( )	Changes the RDN of a leaf entry. This function not supported.
ds_read( )	Queries information on a particular directory entry by name.
ds_receive_result( )	Retrieves the result of an asynchronously executed function. This function not supported.
ds_remove_entry( )	Removes a leaf entry from the CDS namespace.
ds_search( )	Finds entries of interest in a portion of the DIT. This function not supported.
ds_shutdown( )	Discards a workspace.
ds_unbind( )	Unbinds from a directory session.
ds_version( )	Negotiates features of the interface and service.

The XDS Negotiation Sequence

The interface has an initialization and shutdown sequence that permits the negotiation of optional features. This involves the **ds_initialize( )**, **ds_version( )**, and **ds_shutdown( )** functions.

Every application program must first call **ds_initialize( )**, which returns a workspace. This workspace supports the standard directory service package (see “Chapter 10. XDS Class Definitions” on page 189).

The workspace can be extended to support the optional basic directory contents package (see “Chapter 11. Basic Directory Contents Package” on page 219).

Note: ds_version is not required for XDS/XOM over CDS.
These packages are identified by means of OSI object identifiers, and these object identifiers are supplied to **ds_version( )** to incorporate the extensions into the workspace.

After a workspace with the required features is negotiated in this way, the application can use the workspace as required. It can create and manipulate OM objects by using the OM functions, and it can start one or more directory sessions by using **ds_bind( )**.

After completing its tasks, terminating all its directory sessions by using **ds_unbind( )**, and releasing all its OM objects by using **om_delete( )**, the application needs to ensure that resources associated with the interface are freed by calling **ds_shutdown( )**.

It is possible to retain access to service-generated public objects after **ds_shutdown(**) is called, or to start another cycle by calling **ds_initialize( )** if so required by the application design.

The session Parameter

A session binds the XDS/XOM to the CDS namespace. The **session** parameter is passed as the first parameter to most interface functions.

A session is described by an OM object of OM class `DS_C_SESSION`. It is created, and appropriate parameter values can be set with the OM functions. A directory session then starts with **ds_bind( )** and later terminates with **ds_unbind( )**. A session with default parameters can be started by passing the constant `DS_DEFAULT_SESSION` as the `DS_C_SESSION` parameter to **ds_bind(**).

The **ds_bind( )** function must be called before `DS_C_SESSION` can be used as a parameter to any other function in this interface. After **ds_unbind( )** is called, **ds_bind( )** must be called again if another session is to be started.

The interface supports multiple concurrent sessions so that an application implemented as a single process, such as a server in a client/server model, can interact with the directory by using several identities, and a process can interact directly and concurrently with different parts of the directory.

Details of the OM class `DS_C_SESSION` are given in "Chapter 10. XDS Class Definitions" on page 189.

The context Parameter

The context defines the characteristics of the directory interaction that are specific to a particular directory operation; nevertheless, the same characteristics are often used for many operations. Since these parameters are presumed to be relatively static for a given directory user during a particular directory interaction, these parameters are collected into an OM object of OM class `DS_C_CONTEXT`, which is supplied as the second parameter of each directory service request. This reduces the number of parameters passed to each function.

The context includes many administrative details, such as the `CommonArguments` defined in the abstract service, which affect the processing of each directory operation. These details include a number of `ServiceControls`, which allow control over some aspects of the service. The `ServiceControls` include options such as `dontDereferenceAliases`. Each of these is mapped onto an OM attribute in the context (see "Chapter 10. XDS Class Definitions" on page 189).

The effect of passing the *context* parameter is as if its contents were passed as a group of additional parameters for every function call. The value of each component of the context is determined when the interface function is called, and it remains fixed throughout the operation.

All OM attributes in the class DS_C_CONTEXT have default values, some of which are administered locally. The constant DS_DEFAULT_CONTEXT can be passed as the value of the DS_C_CONTEXT parameter to the interface functions, and it has the same effect as a context OM object created with default values. The context must be a private object, unless it is DS_DEFAULT_CONTEXT.

(See “Chapter 10. XDS Class Definitions” on page 189 for detailed specifications of the OM class DS_C_CONTEXT.)

The XDS Function Arguments

The abstract service defines specific parameters for each operation. These are mapped onto corresponding parameters to each interface function, which are also called *input parameters*. Although each service has different parameters, some specific parameters recur in several operations and these are briefly introduced here. (For complete details of these parameters, see “Chapter 10. XDS Class Definitions” on page 189.)

All parameters that are OM objects can generally be supplied to the interface functions as public objects (that is, descriptor lists) or as private objects. Private objects must be created in the workspace that is returned by **ds_initialize( )**. In some cases, constants can be supplied instead of OM objects.

Note: Wherever a function can accept an instance of a particular OM class as the value of a parameter, it also accepts an instance of any subclass of the OM class. For example, most functions have a *name* parameter, which accepts values of OM class *DS_C_NAME*. It is always acceptable to supply an instance of the subclass DS_C_DS_DN as the value of the parameter.

Attribute and Attribute Value Assertion

Each directory attribute is represented in the interface by an OM object of OM class DS_C_ATTRIBUTE. The type of the directory attribute is represented by an OM attribute, DS_ATTRIBUTE_TYPE, within the OM object. The values of the directory attribute are expressed as the values of the OM attribute DS_ATTRIBUTE_VALUES.

The representation of the attribute value depends on the attribute type and is determined as indicated in the following list. The list describes the way in which an application program must supply values to the interface; for example, in the *changes* parameter to **ds_modify_entry( )**. The interface follows the same rules when returning attribute values to the application; for example, in the **ds_read( )** result.

- The first possibility is that the attribute type and the representation of the corresponding values can be defined in a package; for example, the selected attribute types from the standards that are defined in the basic directory contents package in “Chapter 11. Basic Directory Contents Package” on page 219. In this case, attribute values are represented as specified.

- If the attribute type is not known and the value is an ASN.1 simple type such as IntegerType, the representation is the corresponding type specified in "Chapter 12. Information Syntaxes" on page 233.
- If the attribute type is not known and the value is an ASN.1 structured type, the value is represented in the Basic Encoding Rules (BER) with OM syntax String(OM_S_ENCODING_STRING).

Where attribute values have OM syntax String(*), they can be long segmented strings, and the functions **om_read( )** and **om_write( )** need to be used to access them.

An attribute value assertion (AVA) is an assertion about the value of an attribute of an entry, and it can be TRUE, FALSE, or undefined. It consists of an attribute type and a single value. In general, the AVA is TRUE if one of the values of the given attribute in the entry matches the given value. An AVA is represented in the interface by an instance of OM class DS_C_AVA, which is a subclass of DS_C_ATTRIBUTE and can only have one value.

Information used by **ds_add_entry( )** to construct a new directory entry is represented by an OM object of OM class DS_C_ATTRIBUTE_LIST, which contains a single multivalued OM attribute whose values are OM objects of OM class DS_C_ATTRIBUTE.

The selection Parameter

The *selection* parameter of the **ds_read( )** operations tailors its results to obtain just part of the required entry. Information on all attributes, no attributes, or a specific group of attributes can be chosen. Attribute types are always returned, but the attribute values are not necessarily returned.

The value of the parameter is an instance of OM class DS_C_ENTRY_INFO_SELECTION, but one of the constants in the following list can be used in simple cases:

- To verify the existence of an entry for the purported name, use the constant DS_SELECT_NO_ATTRIBUTES.
- To return just the types of all attributes, use the constant DS_SELECT_ALL_TYPES.
- To return the types and values of all attributes, use the constant DS_SELECT_ALL_TYPES_AND_VALUES.

To choose a particular set of attributes, create a new instance of the OM class DS_C_ENTRY_INFO_SELECTION and set the appropriate OM attribute values by using the OM functions.

The name Parameter

Most operations take a

name parameter to specify the target of the operation. The name is represented by an instance of one of the subclasses of the OM class DS_C_NAME. The DCE XDS API defines the subclass DS_C_DS_DN to represent distinguished names and other names.

For directory interrogations, any aliases in the name are dereferenced, unless prohibited by the DS_DONT_DEREFERENCE_ALIASES service control. However, for modify operations, this service control is ignored if set, and aliases are never dereferenced.

RDNs are represented by an instance of one of the subclasses of the OM class `DS_C_RELATIVE_NAME`. The DCE XDS API defines the subclass `DS_C_DS_RDN` to represent RDNs.

XDS Function Call Results

All XDS functions return a `DS_status`, which is the C function result; most return data in an *invoke_id* parameter, which identifies the particular invocation, and the interrogation operations each return data in the *result* parameter. The *invoke_id* and *result* values are returned using pointers that are supplied as parameters of the C function. These three types of function results are introduced in the following subsections.

All OM objects returned by interface functions (results and errors) are private objects in the workspace returned by **ds_initialize( )**.

The invoke_id Parameter

All interface functions that invoke a directory service operation return an *invoke_id* parameter, which is an integer that identifies the particular invocation of an operation. Since asynchronous operations (within the same thread) are not supported, the *invoke_id* return value is no longer relevant for operations. DCE application programmers must still supply this parameter as described in the XDS reference pages, but they should ignore the value returned.

The result Parameter

Directory service interrogation operations return a *result* value only if they succeed. All errors from these operations, including directory access protocol (DAP) errors, are reported in `DS_status` (see “The DS_status Return Value” on page 188), as are errors from all other operations.

The result of an interrogation is returned in a private object whose OM class is appropriate to the particular operation. The format of directory operation results is driven by the abstract service. To simplify processing, the result of a single operation is returned in a single OM object, which corresponds to the abstract result defined in the standards. The components of the result of an operation are represented by OM attributes in the operation’s result object. All information contained in the abstract service result is made available to the application program. The result is inspected using the functions provided in the object management API, **om_get( )**.

Only the interrogation operations produce results, and each type of interrogation has a specific OM class of OM object for its result. These OM classes are as follows (see “Chapter 10. XDS Class Definitions” on page 189 for their definitions):

- `DS_C_COMPARE_RESULT`
- `DS_C_LIST_RESULT`
- `DS_C_READ_RESULT`

The results of the different operations share several common components, including the `CommonResults` defined in the standards (see *The Directory: Abstract Service Definition*, ISO 9594-3, CCITT X.511) by inheriting OM attributes from the superclass *DS_C_COMMON_RESULTS*. An additional common component is the full DN of the target object, after all aliases are dereferenced.

The actual OM class of the result can always be a subclass of that named in order to allow flexibility for extensions. Thus, **om_instance( )** always needs to be used when testing the OM class.

Any attribute values in the result are represented as discussed in "Attribute and Attribute Value Assertion" on page 185.

The DS_status Return Value

Every interface function returns a `DS_status` value, which is either the constant `DS_SUCCESS` or an error. Errors are represented by private objects whose OM class is a subclass of *DS_C_ERROR*. Details of all errors are given in "Chapter 10. XDS Class Definitions" on page 189.

Other results of functions are not valid unless the status result has the value `DS_SUCCESS`.

Synchronous Operations

Since asynchronous use of the interface (within the same thread) is not supported, the value of the `DS_ASYNCHRONOUS` OM attribute in `DS_C_CONTEXT` is always `OM_FALSE`, causing all operations within the same thread to be synchronous.

In synchronous mode, all functions wait until the operation is complete before returning. The thread of control is blocked within the interface after calling a function, and it can use the result immediately after the function returns.

Implementations define a limit on the number of asynchronous operations that can be outstanding at any one time on any one session. The limit is given by the implementation-defined constant `DS_MAX_OUTSTANDING_OPERATIONS`. It always has the value 0 (zero) because asynchronous operations within the same thread are not supported.

All errors occurring during a synchronous request are reported when the function returns. ("Chapter 10. XDS Class Definitions" on page 189 for complete details of error handling.)

The `DS_FILE_DESCRIPTOR` OM attribute of `DS_C_SESSION` is not used by the DCE XDS API and is always set to `DS_NO_VALID_FILE_DESCRIPTOR`.

Chapter 10. XDS Class Definitions

When referring to classes and attributes in the directory service, this guide makes a clear distinction between OM classes and directory classes, and between OM attributes and directory attributes. In both cases, the former is a construct of the closely associated Object Management interface, while the latter is a construct of the directory service where XDS provides access. The terms *object class* and *attribute* indicate the directory constructs, while the phrases *OM class* and *OM attribute* indicate the Object Management constructs.

Note: Not all of the OM objects defined in this section are applicable to XDS/XOM over CDS. One example is **DS_C_SEARCH_RESULT**. It is valid for **ds_search**, but not supported for XDS/XOM over CDS. These OM objects are provided in this documentation for completeness. They should not; however, be needed by any application using XDS/XOM over CDS.

Introduction to OM Classes

This chapter defines, in alphabetical order, the OM classes that constitute the directory service package. This package incorporates the OM classes for the errors that may be returned at the XDS interface. The object identifier associated with this package is

```
{iso(1) identified-organization(3) icd-ecma(0012) member-company(2)
dec(1011) xopen(28) dsp(0)}
```

It takes the following encoding:

```
\x2B\xC\x2\x87\x73\x1C\x0
```

This object identifier is represented by the constant `DS_SERVICE_PKG`.

The object management notation is briefly described in the following text. See “Chapter 12. Information Syntaxes” on page 233 through “Chapter 14. Object Management Package” on page 255 for more information on object management.

Each OM class is described in a separate section, which identifies the OM attributes specific to that OM class. The OM classes and OM attributes for each OM class are listed in alphabetical order. The OM attributes that can be found in an instance of an OM class are those OM attributes specific to that OM class, as well as those inherited from each of its superclasses (see “Chapter 5. XOM Programming” on page 83). The OM class-specific OM attributes are defined in a table. The table indicates the name of each OM attribute, the syntax of each of its values, any restrictions on the length (in bits, octets (bytes), or characters) of each value, any restrictions upon the number of values, and the value, if any, **om_create( )** supplies.

The constants that represent the OM classes and OM attributes in the C binding are defined in the **xds.h(4xds)** header file.

XDS Errors

Errors are reported to the application program by means of `DS_status`, which is a result of every function. (The `DS_status` is *the* function result in the C language binding for most functions.) A function that completes successfully returns the value `DS_SUCCESS`, whereas one that is not successful returns an error. The error is a private object containing details of the problem that occurred. The error constant `DS_NO_WORKSPACE` can be returned by all directory service functions, except **ds_initialize( )**. `DS_NO_WORKSPACE` is returned if **ds_initialize( )** is not invoked before calling any other directory service function.

Errors are classified into ten OM classes. The standards (see *The Directory: Abstract Service Definition*, ISO 9594-3, CCITT X.511) classify errors into eight different groups, as follows:

- Abandoned
- Abandon Failed
- Attribute Error
- Name Error
- Referral
- Security Error
- Service Error
- Update Error

The directory service interface never returns an Abandoned error. The interface also defines three more kinds of errors, as follows:

- `DS_C_LIBRARY_ERROR`
- `DS_C_COMMUNICATIONS_ERROR`
- `DS_C_SYSTEM_ERROR`

Each of these kinds of errors is represented by an OM class. These OM classes are detailed in subsequent sections of this chapter. All of them inherit the OM attribute `DS_PROBLEM` from their superclass *DS_C_ERROR*, which is described in this chapter. The values that `DS_PROBLEM` can take are listed in the relevent subsections of this chapter. For a description of these errors, refer to the *IBM DCE Version 3.1 for AIX and Solaris: Problem Determination Guide*. The error OM classes defined in this chapter are part of the directory service package.

The **ds_bind( )** operation returns a Security Error or a Service Error. All other operations can also return the same errors as **ds_bind( )**. Such errors can arise in the course of following an automatic referral list.

`DS_C_REFERRAL` is not a real error, and it is not a subclass of *DS_C_ERROR*, although it is reported in the same way as a `DS_status` result. A `DS_C_ATTRIBUTE_ERROR`, also not a subclass of *DS_C_ERROR*, is special because it can report several problems at once. Each one is reported in `DS_C_ATTRIBUTE_PROBLEM`, which is a subclass of *DS_C_ERROR*.

OM Class Hierarchy

This section shows the hierarchical organization of the OM classes defined in this chapter and, as a result, shows which OM classes inherit additional OM attributes from their superclasses. In the following list, subclassification is indicated by indentation, and the names of abstract classes are in italics. Thus, for example, the concrete class `DS_C_PRESENTATION_ADDRESS` is an immediate subclass of the abstract class *DS_C_ADDRESS*, which in turn is an immediate subclass of the abstract class *OM_C_OBJECT*. (*OM_C_OBJECT* is defined in “Chapter 14. Object Management Package” on page 255 of this guide.)

OM_C_OBJECT
- `DS_C_ACCESS_POINT`
- *DS_C_ADDRESS*
  - `DS_C_PRESENTATION_ADDRESS`
- `DS_C_ATTRIBUTE`
  - `DS_C_AVA`
  - `DS_C_ENTRY_MOD`
  - `DS_C_FILTER_ITEM`
- `DS_C_ATTRIBUTE_LIST`
  - `DS_C_ENTRY_INFO`
- *DS_C_COMMON_RESULTS*
  - `DS_C_COMPARE_RESULT`
  - `DS_C_LIST_INFO`
  - `DS_C_READ_RESULT`
  - `DS_C_SEARCH_INFO`
- `DS_C_CONTEXT`
- `DS_C_CONTINUATION_REF`
- `DS_C_ENTRY_INFO_SELECTION`
- `DS_C_ENTRY_MOD_LIST`
- *DS_C_ERROR*
- `DS_C_EXT`
- `DS_C_FILTER`
- `DS_C_LIST_INFO_ITEM`
- `DS_C_LIST_RESULT`
- *DS_C_NAME*
  - `DS_C_DS_DN`
- `DS_C_OPERATION_PROGRESS`
- `DS_C_PARTIAL_OUTCOME_QUAL`
- *DS_C_RELATIVE_NAME*
  - `DS_C_DS_RDN`
- `DS_C_SEARCH_RESULT`
- `DS_C_SESSION`

None of the classes in the preceding list are encodable using **om_encode( )** and **om_decode( )**. The application is not permitted to create or modify instances of some OM classes because these OM classes are only returned by the interface and never supplied to it. These OM classes are as follows:

- DS_C_ACCESS POINT
- DS_C_COMPARE_RESULT
- DS_C_CONTINUATION_REF
- All subclasses of *DS_C_ERROR*
- DS_C_LIST_INFO
- DS_C_LIST_INFO_ITEM
- DS_C_LIST_RESULT
- DS_C_OPERATION_PROGRESS
- DS_C_PARTIAL_OUTCOME_QUAL
- DS_C_READ_RESULT
- DS_C_SEARCH_INFO
- DS_C_SEARCH_RESULT

DS_C_ABANDON_FAILED

An instance of OM class DS_C_ABANDON_FAILED reports a problem encountered during an attempt to abandon an operation.

An application is not permitted to create or modify instances of this OM class. An instance of this OM class has the OM attributes of its superclasses, *OM_C_OBJECT* and *DS_C_ERROR*, and no additional OM attributes.

The OM attribute DS_PROBLEM, which is inherited from the superclass *DS_C_ERROR*, identifies the problem. Its value is one of the following:

- DS_E_CANNOT_ABANDON
- DS_E_NO_SUCH_OPERATION
- DS_E_TOO_LATE

A **ds_abandon( )** XDS call always returns a DS_E_TOO_LATE error for the DS_C_ABANDON_FAILED OM class. Refer to "Chapter 9. XDS Interface Description" on page 181 for information on abandoning directory operations.

DS_C_ACCESS_POINT

An instance of OM class DS_C_ACCESS_POINT identifies a particular point at which a DSA can be accessed.

An application is not permitted to create or modify instances of this OM class. An instance of this OM class has the OM attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attributes listed in Table 17.

Table 17. OM Attributes of DS_C_ACCESS_POINT

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_ADDRESS	Object *(DS_C_ ADDRESS)*	—	1	—
DS_AE_TITLE	Object *(DS_C_NAME)*	—	1	—

- DS_ADDRESS

This attribute indicates the address of the DSA to be used when communicating with it.

- DS_AE_TITLE

  This attribute indicates the name of the DSA.

DS_C_ADDRESS

The OM class *DS_C_ADDRESS* represents the address of a particular entity or service, such as a DSA.

It is an abstract class that has the OM attributes of its superclass, *OM_C_OBJECT*, and no other OM attributes.

An address is an unambiguous name, label, or number that identifies the location of the entity or service. All addresses are represented as instances of some subclass of this OM class.

The only subclass defined by the DCE XDS API is DS_C_PRESENTATION_ADDRESS, which is the presentation address of an OSI application entity used for OSI communications with this subclass.

DS_C_ATTRIBUTE

An instance of OM class DS_C_ATTRIBUTE is an attribute of an object, and is thus a component of its directory entry.

An instance of this OM class has the OM attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attributes listed in Table 18.

Table 18. OM Attributes of DS_C_ATTRIBUTE

OM Attribute	**Value Syntax**	**Value Length**	**Value Number**	**Value Initially**
DS_ATTRIBUTE_ TYPE	String(OM_S_ OBJECT_ IDENTIFIER_ STRING)	—	1	—
DS_ATTRIBUTE_ VALUES	Any	—	0 or more	—

- DS_ATTRIBUTE_TYPE

  The attribute type that indicates the class of information given by this attribute.

- DS_ATTRIBUTE_VALUES

  The attribute values. The OM value syntax and the number of values allowed for this OM attribute are determined by the value of the DS_ATTRIBUTE_TYPE OM attribute in accordance with the rules given in “Chapter 9. XDS Interface Description” on page 181.

  If the values of this OM attribute have the syntax String(*), the strings can be long and segmented. For this reason, **om_read( )** and **om_write( )** need to be used to access all String(*) values.

Note: A directory attribute must always have at least one value, although it is acceptable for instances of this OM class not to have any values.

DS_C_ATTRIBUTE_ERROR

An instance of OM class DS_C_ATTRIBUTE_ERROR reports an attribute-related directory service error.

An application is not permitted to create or modify instances of this OM class. An instance of this OM class has the OM attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attributes listed in Table 19.

Table 19. OM Attributes of DS_C_ATTRIBUTE_ERROR

OM Attribute	**Value Syntax**	**Value Length**	**Value Number**	**Value Initially**
DS_OBJECT_ NAME	Object *(DS_C_NAME)*	—	1	—
DS_PROBLEMS	Object(DS_C_ ATTRIBUTE_ PROBLEM)	—	1 or more	—

- DS_OBJECT_NAME

  This attribute contains the name of the directory entry to which the operation is applied when the failure occurs.
- DS_PROBLEMS

  This attribute documents the attribute-related problems encountered. Uniquely, a DS_C_ATTRIBUTE_ERROR can report several problems at once. All problems are related to the preceding object.

DS_C_ATTRIBUTE_LIST

An instance of OM class DS_C_ATTRIBUTE_LIST is a list of directory attributes.

An instance of this OM class has the OM attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attribute listed in Table 20.

Table 20. OM Attribute of DS_C_ATTRIBUTE_LIST

OM Attribute	**Value Syntax**	**Value Length**	**Value Number**	**Value Initially**
DS_ATTRIBUTES	Object(DS_C_ ATTRIBUTE)	—	0 or more	—

- DS_ATTRIBUTES

  This attribute indicates the attributes that constitute a new object's directory entry, or those selected from an existing entry.

DS_C_ATTRIBUTE_PROBLEM

An instance of OM class DS_C_ATTRIBUTE_PROBLEM documents one attribute-related problem encountered while performing an operation as requested on a particular occasion.

An application is not permitted to create or modify instances of this OM class. An instance of this OM class has the OM attributes of its superclasses, *OM_C_OBJECT* and *DS_C_ERROR*, in addition to the OM attributes listed in Table 21 on page 195.

Table 21. OM Attributes of DS_C_ATTRIBUTE_PROBLEM

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_ATTRIBUTE_ TYPE	String(OM_S_ OBJECT_ IDENTIFIER_ STRING)	—	1	—
DS_ATTRIBUTE_ VALUE	Any	—	0 or 1	—

- DS_ATTRIBUTE_TYPE

  This attribute identifies the type of attribute with which the problem is associated.
- DS_ATTRIBUTE_VALUE

  This attribute specifies the attribute value with which the problem is associated. Its syntax is determined by the value of DS_ATTRIBUTE_TYPE. This OM attribute is present if it is necessary to avoid ambiguity.

The OM attribute DS_PROBLEM, which is inherited from the superclass *DS_C_ERROR*, identifies the problem. Its value is one of the following:
- DS_E_ATTRIBUTE_OR_VALUE_EXISTS
- DS_E_CONSTRAINT_VIOLATION
- DS_E_INAPPROP_MATCHING
- DS_E_INVALID_ATTRIBUTE_SYNTAX
- DS_E_NO_SUCH_ATTRIBUTE_OR_VALUE
- DS_E_UNDEFINED_ATTRIBUTE_TYPE

DS_C_AVA

An instance of OM class DS_C_AVA (attribute value assertion) is a proposition concerning the values of a directory entry.

An instance of this OM class has the OM attributes of its superclasses, *OM_C_OBJECT* and DS_C ATTRIBUTE, and no other OM attributes. An additional restriction on this OM class is that there must be exactly one value of the OM attribute DS_ATTRIBUTE_VALUES. The DS_ATTRIBUTE_TYPE remains single valued. The OM value syntax of DS_ATTRIBUTE_VALUES must conform to the rules outlined in “Chapter 9. XDS Interface Description” on page 181.

DS_C_COMMON_RESULTS

The OM class *DS_C_COMMON_RESULTS* comprises results that are returned by, and are common to, the directory interrogation operations.

It is an abstract OM class, which has the OM attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attributes listed in Table 22.

Table 22. OM Attributes of DS_C_COMMON_RESULTS

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_ALIAS_ DEREFERENCED	OM_S_ BOOLEAN	—	1	—
DS_PERFORMER	Object *(DS_C_NAME)*	—	0 or 1	—

- DS_ALIAS_DEREFERENCED

This attribute indicates whether the name of the target object that is passed as a function argument includes an alias that is dereferenced to determine the DN.

- DS_PERFORMER

  When present, this attribute gives the DN of the performer of a particular operation. It can be present when the result is signed, and it holds the name of the DSA that signed the result. The DCE directory service does not support the optional feature of signed results; therefore, this OM attribute is never present.

DS_C_COMMUNICATIONS_ERROR

An instance of OM class DS_C_COMMUNICATIONS_ERROR reports an error occurring in the other OSI services supporting the directory service.

An application is not permitted to create or modify instances of this OM class. An instance of this OM class has the OM attributes of its superclasses, *OM_C_OBJECT* and *DS_C_ERROR*, and no additional OM attributes.

Communications errors include those arising in remote operation, association control, presentation, session, and transport.

The OM attribute DS_PROBLEM, which is inherited from the superclass *DS_C_ERROR*, identifies the problem. Its value is DS_E_COMMUNICATIONS_PROBLEM.

DS_C_COMPARE_RESULT

An instance of OM class DS_C_COMPARE_RESULT comprises the results of a successful call to ds_compare( ).

An application is not permitted to create or modify instances of this OM class. An instance of this OM class has the OM attributes of its superclasses, *OM_C_OBJECT* and *DS_C_COMMON_RESULTS*, in addition to the OM attributes listed in Table 23.

Table 23. OM Attributes of DS_C_COMPARE_RESULT

OM Attribute	**Value Syntax**	**Value Length**	**Value Number**	**Value Initially**
DS_FROM_ENTRY	OM_S_ BOOLEAN	—	1	—
DS_MATCHED	OM_S_ BOOLEAN	—	1	—
DS_OBJECT_NAME	Object *(DS_C_NAME)*	—	0 or 1	—

- DS_FROM_ENTRY

  This attribute indicates whether the assertion is tested against the specified object’s entry, rather than a copy of the entry.

- DS_MATCHED

  This attribute indicates whether the assertion specified as an argument returns the value OM_TRUE. It takes the value OM_TRUE if the values are compared and matched; otherwise, it takes the value OM_FALSE.

- DS_OBJECT_NAME

  This attribute contains the DN of the target object of the operation. It is present if the OM attribute DS_ALIAS_DEREFERENCED, inherited from the superclass *DS_C_COMMON_RESULTS*, is OM_TRUE.

DS_C_CONTEXT

An instance of OM class DS_C_CONTEXT comprises per-operation arguments that are accepted by most of the interface functions.

An instance of this OM class has the OM attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attributes listed in Table 24.

Table 24. OM Attributes of DS_C_CONTEXT

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
Common Arguments				
DS_EXT	Object(DS_C_ EXT)	—	0 or more	NULL
DS_OPERATION_ PROGRESS	Object(DS_C_ OPERATION_ PROGRESS)	—	1	DS_C_OPERATION_NOT_ STARTED
DS_ALIASED_ RDNS	OM_S_ INTEGER	—	0 or 1	0
Service Controls				
DS_CHAINING_ PROHIB	OM_S_ BOOLEAN	—	1	OM_TRUE
DS_DONT_ DEREFERENCE_ ALIASES	OM_S_ BOOLEAN	—	1	OM_FALSE
DS_DONT_ USE_COPY	OM_S_ BOOLEAN	—	1	OM_TRUE
DS_LOCAL_ SCOPE	OM_S_ BOOLEAN	—	1	OM_FALSE
DS_PREFER_ CHAINING	OM_S_ BOOLEAN	—	1	OM_FALSE
DS_PRIORITY	Enum(DS_Priority)	—	1	DS_MEDIUM
DS_SCOPE_ OF_REFERRAL	Enum(DS_ Scope_ of_Referral)	—	0 or 1	DS_COUNTRY
DS_SIZE_ LIMIT	OM_S_ INTEGER	—	0 or 1	—
DS_TIME_ LIMIT	OM_S_ INTEGER	—	0 or 1	—
Local Controls				
DS_ ASYNCHRONOUS	OM_S_ BOOLEAN	—	1	OM_FALSE
DS_AUTOMATIC_ CONTINUATION	OM_S_ BOOLEAN	—	1	OM_TRUE

The context gathers several arguments passed to interface functions, which are presumed to be relatively static for a given directory user during a particular directory interaction. The context is passed as an argument to each function that interrogates or updates the directory. Although it is generally assumed that the context is changed infrequently, the value of each argument can be changed between every operation if required. The DS_ASYNCHRONOUS argument must not be changed. Each argument is represented by one of the OM attributes of the DS_C_CONTEXT OM class.

The context contains the common arguments defined in the standards (see *The Directory: Abstract Service Definition*, ISO 9594-3, CCITT X.511), except that all security information is omitted for reasons discussed in “Chapter 9. XDS Interface Description” on page 181. These are made up of a number of service controls explained in the following text, possible extensions in the DS_EXT OM attribute, and operation progress and alias dereferencing information in the DS_OPERATION_PROGRESS OM attribute. It also contains a number of arguments that provide local control over the interface.

The OM attributes of the DS_C_CONTEXT OM class are as follows:

- Common Arguments
  - DS_EXT

    This attribute represents any future standardized extensions that need to be applied to the directory service operation. The DCE XDS implementation does not evaluate this optional OM attribute.
  - DS_OPERATION_PROGRESS

    This attribute represents the state that the directory service assumes at the start of the operation. This OM attribute normally takes its default value, which is the value DS_OPERATION_NOT_STARTED described in the DS_C_OPERATION_PROGRESS OM class definition.
  - DS_ALIASED_RDNS

    This attribute indicates to the directory service that the object component of the *operation* parameter is created by dereferencing of an alias on an earlier operation attempt. This value is set in the referral response of the previous operation.
- Service Controls
  - DS_CHAINING_PROHIB

    This attribute indicates that chaining and other methods of distributing the request around the directory service are prohibited.
  - DS_DONT_DEREFERENCE_ALIASES

    This attribute indicates that any alias used to identify the target entry of an operation is not dereferenced. This allows interrogation of alias entries. (Aliases are never dereferenced during updates.)
  - DS_DONT_USE_COPY

    This attribute indicates that the request can only be satisfied by accessing directory entries, and not by using copies of entries. This includes both copies maintained in other DSAs by bilateral agreement, and, copies cached locally.
  - DS_LOCAL_SCOPE

    This attribute indicates that the directory request will be satisfied locally. The meaning of this option is configured by an administrator. This option typically restricts the request to a single DSA or DMD.
  - DS_PREFER_CHAINING

    This attribute indicates that chaining is preferred to referrals when necessary. The directory service is not obliged to follow this preference and can return a referral even if it is set.
  - DS_PRIORITY

    This attribute indicates the priority, relative to other directory requests, according to which the directory service attempts to satisfy the request. This is not a guaranteed service since there is no queuing throughout the directory. Its value must be one of the following:
    - DS_LOW
    - DS_MEDIUM
    - DS_HIGH
  - DS_SCOPE_OF_REFERRAL

This attribute indicates the part of the directory to which referrals are limited. This includes referral errors and partial outcome qualifiers. Its value must be one of the following:

  - DS_COUNTRY, meaning DSAs within the country in which the request originates.
  - DS_DMD, meaning DSAs within the DMD in which the request originates.

  DS_SCOPE_OF_REFERRAL is an optional attribute. The lack of this attribute in a DS_C_CONTEXT object indicates that the scope is not limited.

- DS_SIZE_LIMIT

  If present, this attribute indicates the maximum number of objects about which **ds_list( )** or **ds_search( )** needs to return information. If this limit is exceeded, information is returned about exactly this number of objects. The objects that are chosen are not specified because this can depend on the timing of interactions between DSAs, among other reasons.

- DS_TIME_LIMIT

  If present, this attribute indicates the maximum elapsed time, in seconds, within which the service needs to be provided (not the processing time devoted to the request). If this limit is reached, a service error (DS_E_TIME_LIMIT_EXCEEDED) is returned, except for the **ds_list( )** or **ds_search( )** operations, which return an arbitrary selection of the accumulated results.

• Local Controls

- DS_ASYNCHRONOUS (Optional Functionality)

  The interface currently operates synchronously (within the same thread) only, as detailed in “Chapter 9. XDS Interface Description” on page 181. There is only one possible value, as follows:

  - OM_FALSE, meaning that the operation is performed sequentially (synchronously) with the application being blocked until a result or error is returned.

- DS_AUTOMATIC_CONTINUATION

  This attribute indicates the requestor’s requirement for continuation reference handling, including referrals and those in partial outcome qualifiers. The value is one of the following:

  - OM_FALSE, meaning that the interface returns all continuation references to the application program.
  - OM_TRUE, meaning that continuation references are automatically processed, and the subsequent results are returned to the application instead of the continuation references, whenever practical. This is a much simpler option than OM_FALSE unless the application has special requirements.

Note: Continuation references can still be returned to the application if, for example, the relevant DSA cannot be contacted.

Applications can assume that an object of OM class DS_C_CONTEXT, created with default values of all its OM attributes, works with all the interface functions. The DS_DEFAULT_CONTEXT constant can be used as an argument to interface functions instead of creating an OM object with default values.

DS_C_CONTINUATION_REF

An instance of OM class DS_C_CONTINUATION_REF comprises the information that enables a partially completed directory request to be continued; for example, following a referral.

An application is not permitted to create or modify instances of this OM class. An instance of this OM class has the OM attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attributes listed in Table 25.

Table 25. OM Attributes of DS_C_CONTINUATION_REF

OM Attribute	**Value Syntax**	**Value Length**	**Value Number**	**Value Initially**
DS_ACCESS_ POINTS	Object(DS_C_ ACCESS_POINT)	—	1 or more	—
DS_ALIASED_ RDNS	OM_S_INTEGER	—	1	—
DS_OPERATION_ PROGRESS	Object(DS_C_ OPERATION_ PROGRESS)	—	1	—
DS_RDNS_ RESOLVED	OM_S_INTEGER	—	0 or 1	—
DS_TARGET_ OBJECT	Object *(DS_C_NAME)*	—	1	—

- DS_ACCESS_POINTS

  This attribute indicates the names and presentation addresses of the DSAs from where the directory request is continued.
- DS_ALIASED_RDNS

  This attribute indicates how many (if any) of the RDNs in the target name are produced by dereferencing an alias. Its value is 0 (zero) if no aliases are dereferenced. This value needs to be used in the DS_C_CONTEXT of any continued operation.
- DS_OPERATION_PROGRESS

  This attribute indicates the state at which the directory request must be continued. This value needs to be used in the DS_C_CONTEXT of any continued operation.
- DS_RDNS_RESOLVED

  This attribute indicates the number of RDNs in the supplied object name that are resolved (using internal references), and not just assumed to be correct (using cross-references).
- DS_TARGET_OBJECT

  This attribute indicates the name of the object upon which the continuation must focus.

DS_C_DS_DN

An instance of OM class DS_C_DS_DN represents a name of a directory object.

An instance of this OM class has the OM attributes of its superclasses, *OM_C_OBJECT* and *DS_C_NAME*, in addition to the OM attribute listed in Table 26 on page 201.

Table 26. OM Attribute of DS_C_DS_DN

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_RDNS	Object(DS_C_DS_ RDN)	—	0 or more	—

- DS_RDNS

  This attribute indicates the sequence of RDNs that define the path through the DIT from its root to the object that the DS_C_DS_DN indicates. The DS_C_DS_DN of the root of the directory is the null name (no DS_RDNS values). The order of the values is significant; the first value is closest to the root, and the last value is the RDN of the object.

DS_C_DS_RDN

An instance of OM class DS_C_DS_RDN is a relative distinguished name. An RDN uniquely identifies an immediate subordinate of an object whose entry is displayed in the DIT.

An instance of this OM class has the OM attributes of its superclasses, *OM_C_OBJECT* and *DS_C_RELATIVE_NAME*, in addition to the OM attribute listed in Table 27.

Table 27. OM Attribute of DS_C_DS_RDN

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_AVAS	Object(DS_C_AVA)	—	1 or more	—

- DS_AVAS

  This attribute indicates the DS_AVAS that are marked by the DIB as components of the object's RDN. The assertion is TRUE of the object but not of any of its siblings, and the attribute type and value are displayed in the object's directory entry. The order of the DS_AVAS is not significant.

DS_C_ENTRY_INFO

An instance of OM class DS_C_ENTRY_INFO contains selected information from a single directory entry.

An instance of this OM class has the OM attributes of its superclasses, *OM_C_OBJECT* and DS_C_ATTRIBUTE_LIST, in addition to the OM attributes listed in Table 28.

Table 28. OM Attributes of DS_C_ENTRY_INFO

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_FROM_ENTRY	OM_S_ BOOLEAN	—	1	—
DS_OBJECT_NAME	Object *(DS_C_ NAME)*	—	1	—

The OM attribute DS_ATTRIBUTES is inherited from the superclass DS_C_ATTRIBUTE_LIST. It contains the information extracted from the directory entry of the target object. The type of each attribute requested and located is indicated in the list as are its values, if types and values are requested.

The OM class-specific OM attributes are as follows:

- DS_FROM_ENTRY

  This attribute indicates whether the information is extracted from the specified object's entry, rather than from a copy of the entry.
- DS_OBJECT_NAME

  This attribute contains the object's DN.

DS_C_ENTRY_INFO_SELECTION

An instance of OM class DS_C_ENTRY_INFO_SELECTION identifies the information to be extracted from a directory entry.

An instance of this OM class has the OM attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attributes listed in Table 29.

Table 29. OM Attributes of DS_C_ENTRY_INFO_SELECTION

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_ALL_ATTRIBUTES	OM_S_ BOOLEAN	—	1	OM_TRUE
DS_ ATTRIBUTES_ SELECTED	String(OM_S_OBJECT_ IDENTIFIER_STRING)	—	0 or more	–
DS_INFO_TYPE	Enum(DS_Information_ Type)	—	1	DS_TYPES_ AND_VALUES

- DS_ALL_ATTRIBUTES

  This attribute indicates which attributes are relevant. It can take one of the following values:
  - OM_FALSE, meaning that information is only requested on those attributes that are listed in the OM attribute DS_ATTRIBUTES_SELECTED.
  - OM_TRUE, meaning that information is requested on all attributes in the directory entry. Any values of the OM attribute DS_ATTRIBUTES_SELECTED are ignored in this case.
- DS_ATTRIBUTES_SELECTED

  This attribute lists the types of attributes in the entry from which information will be extracted. The value of this OM attribute is used only if the value of DS_ALL_ATTRIBUTES is OM_FALSE. If an empty list is supplied, no attribute data is returned that could be used to verify the existence of an entry for a DN.
- DS_INFO_TYPE

  This attribute identifies the information that will be extracted from each attribute identified. It must take one of the following values:
  - DS_TYPES_ONLY, meaning that only the attribute types of the selected attributes in the entry are returned.
  - DS_TYPES_AND_VALUES, meaning that both the attribute types and the attribute values of the selected attributes in the entry are returned.

DS_C_ENTRY_MOD

An instance of OM class DS_C_ENTRY_MOD describes a single modification to a specified attribute of a directory entry.

An instance of this OM class has the OM attributes of its superclasses, *OM_C_OBJECT* and DS_C_ATTRIBUTE, in addition to the OM attribute listed in Table 30 on page 203.

Table 30. OM Attribute of DS_C_ENTRY_MOD

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_MOD_TYPE	Enum(DS_ Modification_ Type)	—	1	DS_ADD_ ATTRIBUTE

The attribute type to be modified, and the associated values, are specified in the OM attributes DS_ATTRIBUTE_TYPE and DS_ATTRIBUTE_VALUES that are inherited from the DS_C_ATTRIBUTE superclass.

- DS_MOD_TYPE

  This attribute identifies the type of modification. It must have one of the following values:
  - DS_ADD_ATTRIBUTE, meaning that the specified attribute is absent and will be added with the specified values.
  - DS_ADD_VALUES, meaning that the specified attribute is present and that one or more specified values will be added to it.
  - DS_REMOVE_ATTRIBUTE, meaning that the specified attribute is present and will be removed. Any values present in the OM attribute DS_ATTRIBUTE_VALUES are ignored.
  - DS_REMOVE_VALUES, meaning that the specified attribute is present and that one or more specified values will be removed from it.

DS_C_ENTRY_MOD_LIST

An instance of OM class DS_C_ENTRY_MOD_LIST comprises a sequence of changes to be made to a directory entry.

An instance of this OM class has the OM attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attribute listed in Table 31.

Table 31. OM Attribute of DS_C_ENTRY_MOD_LIST

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_CHANGES	Object(DS_C_ ENTRY_MOD)	—	1 or more	—

- DS_CHANGES

  This attribute identifies the modifications to be made (in the order specified) to the directory entry of the specified object.

DS_C_ERROR

The OM class *DS_C_ERROR* comprises the parameters common to all errors.

It is an abstract OM class with the OM attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attribute listed in Table 32.

Table 32. OM Attribute of DS_C_ERROR

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_PROBLEM	Enum(DS_Problem)	—	1	—

Details of errors are returned in an instance of a subclass of this OM class. Each such subclass represents a particular kind of error, and is one of the following:

- DS_C_ABANDON_FAILED
- DS_C_ATTRIBUTE_PROBLEM
- DS_C_COMMUNICATIONS_ERROR
- DS_C_LIBRARY_ERROR
- DS_C_NAME_ERROR
- DS_C_SECURITY_ERROR
- DS_C_SERVICE_ERROR
- DS_C_SYSTEM_ERROR
- DS_C_UPDATE_ERROR

A number of possible values are defined for these subclasses. DCE XDS does not return other values for error conditions described in this chapter. Information on system errors can be found in “DS_C_SYSTEM_ERROR” on page 217. The following is a list of the error values. Each error OM class section defines the possible error values associated with that class. For a description of the errors, refer to the *IBM DCE Version 3.1 for AIX and Solaris: Problem Determination Guide*.

- DS_E_ADMIN_LIMIT_EXCEEDED
- DS_E_AFFECTS_MULTIPLE_DSAS
- DS_E_ALIAS_DEREFERENCING_PROBLEM
- DS_E_ALIAS_PROBLEM
- DS_E_ATTRIBUTE_OR_VALUE_EXISTS
- DS_E_BAD_ARGUMENT
- DS_E_BAD_CLASS
- DS_E_BAD_CONTEXT
- DS_E_BAD_NAME
- DS_E_BAD_SESSION
- DS_E_BAD_WORKSPACE
- DS_E_BUSY
- DS_E_CANNOT_ABANDON
- DS_E_CHAINING_REQUIRED
- DS_E_COMMUNICATIONS_PROBLEM
- DS_E_CONSTRAINT_VIOLATION
- DS_E_DIT_ERROR
- DS_E_ENTRY_EXISTS
- DS_E_INAPPROP_AUTHENTICATION
- DS_E_INAPPROP_MATCHING
- DS_E_INSUFFICIENT_ACCESS_RIGHTS
- DS_E_INVALID_ATTRIBUTE_SYNTAX
- DS_E_INVALID_ATTRIBUTE_VALUE
- DS_E_INVALID_CREDENTIALS
- DS_E_INVALID_REF
- DS_E_INVALID_SIGNATURE
- DS_E_LOOP_DETECTED
- DS_E_MISCELLANEOUS

- DS_E_MISSING_TYPE
- DS_E_MIXED_SYNCHRONOUS
- DS_E_NAMING_VIOLATION
- DS_E_NO_INFO
- DS_E_NO_SUCH_ATTRIBUTE_OR_VALUE
- DS_E_NO_SUCH_OBJECT
- DS_E_NO_SUCH_OPERATION
- DS_E_NOT_ALLOWED_ON_NON_LEAF
- DS_E_NOT_ALLOWED_ON_RDN
- DS_E_NOT_SUPPORTED
- DS_E_OBJECT_CLASS_MOD_PROHIB
- DS_E_OBJECT_CLASS_VIOLATION
- DS_E_OUT_OF_SCOPE
- DS_E_PROTECTION_REQUIRED
- DS_E_TIME_LIMIT_EXCEEDED
- DS_E_TOO_LATE
- DS_E_TOO_MANY_OPERATIONS
- DS_E_TOO_MANY_SESSIONS
- DS_E_UNABLE_TO_PROCEED
- DS_E_UNAVAILABLE
- DS_E_UNAVAILABLE_CRIT_EXT
- DS_E_UNDEFINED_ATTRIBUTE_TYPE
- DS_E_UNWILLING_TO_PERFORM

DS_C_EXT

An instance of OM class DS_C_EXT indicates that a standardized extension to the directory service is outlined in the standards. Such extensions will only be standardized in post-1988 versions of the standards. Therefore, this OM class is not used by the XDS API and is only included for X/Open conformance purposes.

An instance of this OM class has the OM attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attributes listed in Table 33.

Table 33. OM Attributes of DS_C_EXT

OM Attribute	**Value Syntax**	**Value Length**	**Value Number**	**Value Initially**
DS_CRIT	OM_S_BOOLEAN	—	1	OM_FALSE
DS_IDENT	OM_S_INTEGER	—	1	—
DS_ITEM_ PARAMETERS	Any	—	1	—

- DS_CRIT

  This attribute must have one of the following values:
  - OM_FALSE, meaning that the originator permits the operation to be performed even if the extension is not available.
  - OM_TRUE, meaning that the originator mandates that the extended operation be performed. If the extended operation is not performed, an error is reported.
- DS_IDENT

This attribute identifies the service extension.

- DS_ITEM_PARAMETERS

  This OM attribute supplies the parameters of the extension. Its syntax is determined by the value of DS_IDENT.

DS_C_FILTER

An instance of OM class DS_C_FILTER is used to select or reject an object on the basis of information in its directory entry. At any point in time, an attribute filter has a value relative to every object. The value is FALSE, TRUE, or undefined. The object is selected if, and only if, the filter's value is TRUE.

An instance of this OM class has the OM attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attributes listed in Table 34.

Table 34. OM Attributes of DS_C_FILTER

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_FILTER_ITEMS	Object(DS_C_ FILTER_ITEM)	—	0 or more	—
DS_FILTERS	Object(DS_C_ FILTER)	—	0 or more	—
DS_FILTER_TYPE	Enum(DS_Filter_ Type)	—	1	DS_AND

A *filter* is a collection of less elaborate filters and elementary DS_FILTER_ITEMS, together with a Boolean operation. The filter value is undefined if, and only if, all the component DS_FILTERS and DS_FILTER_ITEMS are undefined. Otherwise, the filter has a Boolean value with respect to any directory entry, which can be determined by evaluating each of the nested components and combining their values using the Boolean operation. The components whose values are undefined are ignored.

- DS_FILTER_ITEMS

  This attribute is a collection of assertions, each relating to just one attribute of a directory entry.

- DS_FILTERS

  This attribute is a collection of simpler filters.

- DS_FILTER_TYPE

  This attribute is the filter's type. It can have any of the following values:

  – DS_AND, meaning that the filter is the logical conjunction of its components. The filter is TRUE unless any of the nested filters or filter items is FALSE. If there are no nested components, the filter is TRUE.

  – DS_OR, meaning that the filter is the logical disjunction of its components. The filter is FALSE unless any of the nested filters or filter items is TRUE. If there are no nested components, the filter is FALSE.

  – DS_NOT, meaning that the result of this filter is reversed. There must be exactly one nested filter or filter item. The filter is TRUE if the enclosed filter or filter item is FALSE, and it is FALSE if the enclosed filter or filter item is TRUE.

DS_C_FILTER_ITEM

An instance of OM class DS_C_FILTER_ITEM is a component of DS_C_FILTER. It is an assertion about the existence or values of a single attribute type in a directory entry.

An instance of this OM class has the OM attributes of its superclasses, *OM_C_OBJECT* and DS_C_ATTRIBUTE, in addition to the OM attributes listed in Table 35.

Table 35. OM Attributes of DS_C_FILTER_ITEM

OM Attribute	**Value Syntax**	**Value Length**	**Value Number**	**Value Initially**
DS_FILTER_ ITEM_TYPE	Enum(DS_Filter_ Item_Type)	—	1	—
DS_FINAL_ SUBSTRING	String(*)	1 or more	0 or 1	—
DS_INITIAL_ SUBSTRING	String(*)	1 or more	0 or 1	—

Note: OM attributes DS_ATTRIBUTE_TYPE and DS_ATTRIBUTE_VALUES are inherited from the superclass DS_C_ATTRIBUTE.

The value of the filter item is undefined in the following cases:

- The DS_ATTRIBUTE_TYPE is not known.
- None of the DS_ATTRIBUTE_VALUES conform to the attribute syntax defined for that attribute type.
- The DS_FILTER_ITEM_TYPE uses a matching rule that is not defined for the attribute syntax.

Access control restrictions can also cause the value to be undefined.

- DS_FILTER_ITEM_TYPE

  This attribute identifies the type of filter item and, thus, the nature of the filter. The filter item can adopt any of the following values:

  – DS_APPROXIMATE_MATCH, meaning that the filter is TRUE if the directory entry contains at least one value of the specified type that is approximately equal to that specified (the meaning of ″approximately equal″ is implementation dependent); otherwise, the filter is FALSE.

    Rules for approximate matching are defined locally. For example, an approximate match may take into account spelling variations or employ phonetic comparison rules. In the absence of any such capabilities, a DSA needs to treat an approximate match as a test for equality. DCE GDS supports phonetic comparisons. There must be exactly one value of the OM attribute DS_ATTRIBUTE_VALUES.

  – DS_EQUALITY, meaning that the filter is TRUE if the entry contains at least one value of the specified type that is equal to the value specified, according to the equality matching rule in force; otherwise, the filter is FALSE. There must be exactly one value of the OM attribute DS_ATTRIBUTE_VALUES.

  – DS_GREATER_OR_EQUAL, meaning that the filter item is TRUE if, and only if, at least one value of the attribute is greater than or equal to the supplied value. There must be exactly one value of the OM attribute DS_ATTRIBUTE_VALUES.

  – DS_LESS_OR_EQUAL, meaning that the filter item is TRUE if, and only if, at least one value of the attribute is less than or equal to the supplied value. There must be exactly one value of the OM attribute DS_ATTRIBUTE_VALUES.

  – DS_PRESENT, meaning that the filter is TRUE if the entry contains an attribute of the specified type; otherwise, it is FALSE.

    Any values of the OM attribute DS_ATTRIBUTE_VALUES are ignored.

  – DS_SUBSTRINGS, meaning that the filter is TRUE if the entry contains at least one value of the specified attribute type that contains all of the specified substrings in the given order; otherwise, the filter is FALSE.

Any number of substrings can be given as values of the OM attribute DS_ATTRIBUTE_VALUES. Similarly, no substrings can be specified. There can also be a substring in DS_INITIAL_SUBSTRING or DS_FINAL_SUBSTRING, or both. The substrings do not overlap, but they can be separated from each other or from the ends of the attribute value by zero or more string elements. However, at least one attribute of type DS_ATTRIBUTE_VALUES, DS_INITIAL_SUBSTRING, or DS_FINAL_SUBSTRING must exist.

- DS_FINAL_SUBSTRING

  If present, this attribute is the substring that will match the final part of an attribute value in the entry. This attribute can only exist if the DS_FILTER_ITEM_TYPE is equal to DS_SUBSTRINGS.
- DS_INITIAL_SUBSTRING

  If present, this attribute is the substring that will match the initial part of an attribute value in the entry.

DS_C_LIBRARY_ERROR

An instance of OM class DS_C_LIBRARY_ERROR reports an error detected by the interface function library.

An application is not permitted to create or modify instances of this OM class. An instance of this OM class has the OM attributes of its superclasses, *OM_C_OBJECT* and *DS_C_ERROR*, and no additional OM attributes.

Each function has several possible errors that can be detected by the library itself and that are returned directly by the subroutine. These errors occur when the library itself is incapable of performing an action, submitting a service request, or deciphering a response from the directory service.

The OM attribute DS_PROBLEM, which is inherited from the superclass *DS_C_ERROR*, identifies the particular library error that occurred. (In reference pages, the ERRORS section of each function description lists the errors that the respective function can return.) Its value is one of the following:

- DS_E_BAD_ARGUMENT
- DS_E_BAD_CLASS
- DS_E_BAD_CONTEXT
- DS_E_BAD_NAME
- DS_E_BAD_SESSION
- DS_E_MISCELLANEOUS
- DS_E_MISSING_TYPE
- DS_E_MIXED_SYNCHRONOUS
- DS_E_NOT_SUPPORTED
- DS_E_TOO_MANY_OPERATIONS
- DS_E_TOO_MANY_SESSIONS

DS_C_LIST_INFO

An instance of OM class DS_C_LIST_INFO is part of the results of **ds_list( )**.

An application is not permitted to create or modify instances of this OM class. An instance of this OM class has the OM attributes of its superclasses, *OM_C_OBJECT* and *DS_C_COMMON_RESULTS*, in addition to the OM attributes listed in Table 36.

Table 36. OM Attributes of DS_C_LIST_INFO

OM Attribute	**Value Syntax**	**Value Length**	**Value Number**	**Value Initially**
DS_OBJECT_ NAME	Object *(DS_C_NAME)*	—	0 or 1	—
DS_PARTIAL_ OUTCOME_QUAL	Object(DS_C_ PARTIAL_ OUTCOME_QUAL)	—	0 or 1	—
DS_ SUBORDINATES	Object(DS_C_ LIST_INFO_ITEM)	—	0 or more	—

- DS_OBJECT_NAME

  This attribute is the DN of the target object of the operation. It is present if the OM attribute DS_ALIAS_DEREFERENCED, inherited from the superclass *DS_C_COMMON_RESULTS*, is OM_TRUE.
- DS_PARTIAL_OUTCOME_QUAL

  This OM attribute value is present if the list of subordinates is incomplete. The DSA or DSAs that provided this list did not complete the search for some reason. The partial outcome qualifier contains details of why the search is not completed, and which areas of the directory have not been searched.
- DS_SUBORDINATES

  This attribute contains information about zero or more subordinate objects identified by **ds_list( )**.

DS_C_LIST_INFO_ITEM

An instance of OM class DS_C_LIST_INFO_ITEM comprises details returned by **ds_list( )** of a single subordinate object.

An application is not permitted to create or modify instances of this OM class. An instance of this OM class has the OM attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attributes listed in Table 37.

Table 37. OM Attributes of DS_C_LIST_INFO_ITEM

OM Attribute	**Value Syntax**	**Value Length**	**Value Number**	**Value Initially**
DS_ALIAS_ENTRY	OM_S_ BOOLEAN	—	1	—
DS_FROM_ENTRY	OM_S_ BOOLEAN	—	1	—
DS_RDN	Object *(DS_C_ RELATIVE_ NAME)*	—	1	—

- DS_ALIAS_ENTRY

  This attribute indicates whether the object is an alias.
- DS_FROM_ENTRY

  This attribute indicates whether information about the object was obtained directly from its directory entry, rather than from a copy of the entry.
- DS_RDN

  This attribute contains the RDN of the object. If this is the name of an alias entry, as indicated by DS_ALIAS_ENTRY, it is not dereferenced.

DS_C_LIST_RESULT

An instance of OM class DS_C_LIST_RESULT comprises the results of a successful call to **ds_list( )**.

An application is not permitted to create or modify instances of this OM class. An instance of this OM class has the OM attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attributes listed in Table 38.

Table 38. OM Attributes of DS_C_LIST_RESULT

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_LIST_INFO	Object(DS_C_ LIST_INFO)	—	0 or 1	—
DS_ UNCORRELATED_ LIST_INFO	Object(DS_C_ LIST_ RESULT)	—	0 or more	—

Note: No instance contains values of both OM attributes.

- DS_LIST_INFO

  This attribute contains the full results of ds_list( ), or just part of them.
- DS_UNCORRELATED_LIST_INFO

  When the DUA requests a protection request of *signed*, the information returned can comprise a number of sets of results originating from, and signed by, different components of the directory. Implementations can reflect this structure by nesting DS_LIST_RESULT OM objects as values of this OM attribute. Alternatively, they can collapse all results into a single value of the OM attribute DS_LIST_INFO. The DCE directory service does not support the optional feature of signed results; therefore, this OM attribute is never present.

DS_C_NAME

The OM class *DS_C_NAME* represents a name of an object in the directory, or a part of such a name.

It is an abstract class that has the attributes of its superclass, *OM_C_OBJECT*, and no other OM attributes.

A name uniquely distinguishes the object from all other objects whose entries are displayed in the DIT. However, an object can have more than one name; that is, a name need not be unique. A DN is unique; there are no other DNs that identify the same object. An RDN is part of a name and only distinguishes the object from others that are its siblings.

Most of the interface functions take a *name* parameter, the value of which must be an instance of one of the subclasses of this OM class. Thus, this OM class is useful for amalgamating all possible representations of names.

The DCE XDS implementation defines one subclass of this OM class and, thus, a single representation for names; that is, DS_C_DS_DN, which provides a representation for names, including DNs.

DS_C_NAME_ERROR

An instance of OM class DS_C_NAME_ERROR reports a name-related directory service error.

An application is not permitted to create or modify instances of this OM class. An instance of this OM class has the OM attributes of its superclasses, *OM_C_OBJECT* and *DS_C_ERROR*, in addition to the OM attribute listed in Table 39.

Table 39. OM Attribute of DS_C_NAME_ERROR

OM Attribute	**Value Syntax**	**Value Length**	**Value Number**	**Value Initially**
DS_MATCHED	Object *(DS_C_NAME)*	—	1	—

- DS_MATCHED

  This attribute identifies the initial part (up to, but excluding, the first RDN that is unrecognized) of the name that is supplied, or of the name resulting from dereferencing an alias. It names the lowest entry (object or alias) in the DIT that is matched.

The OM attribute DS_PROBLEM, which is inherited from the superclass *DS_C_ERROR*, identifies the cause of the failure. Its value is one of the following:

- DS_E_ALIAS_DEREFERENCING_PROBLEM
- DS_E_ALIAS_PROBLEM
- DS_E_INVALID_ATTRIBUTE_VALUE
- DS_E_NO_SUCH_OBJECT

DS_C_OPERATION_PROGRESS

An instance of OM class DS_C_OPERATION_PROGRESS specifies the progress or processing state of a directory request.

An application is not permitted to create or modify instances of this OM class. An instance of this OM class has the OM attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attributes listed in Table 40.

Table 40. OM Attributes of DS_C_OPERATION_PROGRESS

OM Attribute	**Value Syntax**	**Value Length**	**Value Number**	**Value Initially**
DS_NAME_ RESOLUTION_ PHASE	Enum(DS_Name_ Resolution_Phase)	—	1	—
DS_NEXT_ RDN_TO_BE RESOLVED	OM_S_INTEGER	—	0 or 1	—

The target name mentioned as follows is the name upon which processing of the directory request is currently focused.

- DS_NAME_RESOLUTION_PHASE

  This attribute indicates what phase is reached in handling the target name. It must have one of the following values:

  – DS_COMPLETED, meaning that the DSA holding the target object is reached.

  – DS_NOT_STARTED, meaning that so far a DSA is not reached with a naming context containing the initial RDNs of the name.

  - DS_PROCEEDING, meaning that the initial part of the name has been recognized, although the DSA holding the target object has not yet been reached.
- DS_NEXT_RDN_TO_BE_RESOLVED

  This attribute indicates to the DSA which of the RDNs in the target name is next to be resolved. It takes the form of an integer in the range from 1 to the number of RDNs in the name. This OM attribute only has a value if the value of DS_NAME_RESOLUTION_PHASE is DS_PROCEEDING.

The constant DS_OPERATION_NOT_STARTED can be used in the DS_C_CONTEXT of an operation instead of an instance of this OM class.

DS_C_PARTIAL_OUTCOME_QUAL

An instance of OM class DS_C_PARTIAL_OUTCOME_QUAL explains to what extent the results of a call to **ds_list( )** or **ds_search( )** are incomplete and why.

An application is not permitted to create or modify instances of this OM class. An instance of this OM class has the OM attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attributes listed in Table 41.

Table 41. OM Attributes of a DS_C_PARTIAL_OUTCOME_QUAL

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_LIMIT_ PROBLEM	Enum(DS_Limit_ Problem)	—	1	—
DS_ UNAVAILABLE_ CRIT_EXT	OM_S_ BOOLEAN	—	1	—
DS_ UNEXPLORED	Object(DS_C_ CONTINUATION_ REF)	—	0 or more	—

- DS_LIMIT_PROBLEM

  This attribute explains fully or partly why the results are incomplete. It can have one of the following values:
  - DS_ADMIN_LIMIT_EXCEEDED, meaning that an administrative limit is reached.
  - DS_NO_LIMIT_EXCEEDED, meaning that there is no limit problem.
  - DS_SIZE_LIMIT_EXCEEDED, meaning that the maximum number of objects specified as a service control is reached.
  - DS_TIME_LIMIT_EXCEEDED, meaning that the maximum number of seconds specified as a service control is reached.
- DS_UNAVAILABLE_CRIT_EXT

  If OM_TRUE, this attribute indicates that some part of the directory service cannot provide a requested critical service extension. The user requested one or more standard service extensions by including values of the OM attribute DS_EXT in the DS_C_CONTEXT supplied for the operation. Furthermore, the user indicated that some of these extensions are essential by setting the OM attribute DS_CRIT in the extension to OM_TRUE. Some of the critical extensions cannot be performed by one particular DSA or by a number of DSAs. In general, it is not possible to determine which DSA could not perform which particular extension.
- DS_UNEXPLORED

  This attribute identifies any regions of the directory that are left unexplored in such a way that the directory request can be continued. Only continuation references within the scope specified by the DS_SCOPE_OF_REFERRAL service control are included.

DS_C_PRESENTATION_ADDRESS

An instance of OM class DS_C_PRESENTATION_ADDRESS is a presentation address of an OSI application entity, which is used for OSI communications with this instance.

An instance of this OM class has the OM attributes of its superclasses, *OM_C_OBJECT* and *DS_C_ADDRESS*, in addition to the OM attributes listed in Table 42.

Table 42. OM Attributes of DS_C_PRESENTATION_ADDRESS

OM Attribute	**Value Syntax**	**Value Length**	**Value Number**	**Value Initially**
DS_N_ ADDRESSES	String(OM_S_ OCTET_STRING)	—	1 or more	—
DS_P_SELECTOR	String(OM_S_ OCTET_STRING)	—	0 or 1	—
DS_S_SELECTOR	String(OM_S_ OCTET_STRING)	—	0 or 1	—
DS_T_SELECTOR	String(OM_S_ OCTET_STRING)	—	0 or 1	—

- DS_N_ADDRESSES
  This attribute is the network addresses of the application entity.
- DS_P_SELECTOR
  This attribute is the presentation selector.
- DS_S_SELECTOR
  This attribute is the session selector.
- DS_T_SELECTOR
  This attribute is the transport selector.

DS_C_READ_RESULT

An instance of OM class DS_C_READ_RESULT comprises the result of a successful call to **ds_read( )**. An application is not permitted to create or modify instances of this OM class. An instance of this OM class has the OM attributes of its superclasses, *OM_C_OBJECT* and *DS_C_COMMON_RESULTS*, in addition to the OM attribute listed in Table 43.

Table 43. OM Attribute of DS_C_READ_RESULT

OM Attribute	**Value Syntax**	**Value Length**	**Value Number**	**Value Initially**
DS_ENTRY	Object(DS_C_ ENTRY_INFO)	—	1	—

- DS_ENTRY
  This attribute contains the information extracted from the directory entry of the target object.

DS_C_REFERRAL

An instance of OM class DS_C_REFERRAL reports failure to perform an operation and redirects the requestor to one or more access points better equipped to perform the operation.

An application is not permitted to create or modify instances of this OM class. An instance of this OM class has the OM attributes of its superclasses, *OM_C_OBJECT* and DS_C_CONTINUATION_REF, and no additional OM attributes.

The referral is a continuation reference by means of which the operation can progress.

DS_C_RELATIVE_NAME

The OM class *DS_C_RELATIVE_NAME* represents the RDNs of objects in the directory. It is an abstract class, which has the attributes of its superclass, *OM_C_OBJECT*, and no other OM attributes.

An RDN is part of a name, and only distinguishes the object from others that are its siblings. This OM class is used to accumulate all possible representations of RDNs. An argument of interface functions that is an RDN, or an OM attribute value that is an RDN is an instance of one of the subclasses of this OM class.

The DCE XDS API defines one subclass of this OM class, and, thus, a single representation for RDNs; that is, DS_C_DS_RDN, which provides a representation for RDNs.

DS_C_SEARCH_INFO

An instance of OM class DS_C_SEARCH_INFO is part of the result of **ds_search( )**.

An application is not permitted to create or modify instances of this OM class. An instance of this OM class has the OM attributes of its superclasses, *OM_C_OBJECT* and *DS_C_COMMON_RESULTS*, in addition to the OM attributes listed in Table 44.

Table 44. OM Attributes of DS_C_SEARCH_INFO

OM Attribute	**Value Syntax**	**Value Length**	**Value Number**	**Value Initially**
DS_ENTRIES	Object(DS_C_ ETNRY_INFO)	—	0 or more	—
DS_OBJECT_ NAME	Object *(DS_C_NAME)*	—	0 or 1	—
DS_PARTIAL_ OUTCOME_QUAL	Object(DS_C_ PARTIAL_ OUTCOME_QUAL)	—	0 or 1	—

- DS_ENTRIES

  This attribute contains information about zero or more objects found by **ds_search( )** that matched the given selection criteria.
- DS_OBJECT_NAME

  This attribute contains the DN of the target object of the operation. It is present if the OM attribute DS_ALIAS_DEREFERENCED, inherited from the superclass *DS_C_COMMON_RESULTS*, is OM_TRUE.
- DS_PARTIAL_OUTCOME_QUAL

  This OM attribute value is only present if the list of entries is incomplete. The DSA or DSAs that provided this list did not complete the search for some reason. The partial outcome qualifier contains details of why the search was not completed and which areas of the directory were not searched.

DS_C_SEARCH_RESULT

An instance of OM class DS_C_SEARCH_RESULT comprises the result of a successful call to **ds_search( )**.

An application is not permitted to create or modify instances of this OM class. An instance of this OM class has the OM attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attributes listed in Table 45.

Table 45. OM Attributes of DS_C_SEARCH_RESULT

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_SEARCH_INFO	Object(DS_C_ SEARCH_INFO)	—	0 or 1	—
DS_ UNCORRELATED_ SEARCH_INFO	Object(DS_C_ SEARCH_RESULT)	—	0 or more	—

Note: No instance contains values of both OM attributes.

- DS_SEARCH_INFO

  This attribute contains the full result of ds_search( ), or part of the result.
- DS_UNCORRELATED_SEARCH_INFO

  When the DUA requests a protection request of *signed*, the information returned can comprise a number of sets of results originating from and signed by different components of the directory service. Implementations can reflect this structure by nesting DS_C_SEARCH_RESULT OM objects as values of this OM attribute. Alternatively, they can collapse all results into a single value of the OM attribute DS_SEARCH_INFO. The DCE directory service does not support the optional feature of signed results; therefore, this OM attribute is never present.

DS_C_SECURITY_ERROR

An instance of OM class DS_C_SECURITY_ERROR reports a security-related directory service error.

An application is not permitted to create or modify instances of this OM class. An instance of this OM class has the OM attributes of its superclasses, *OM_C_OBJECT* and *DS_C_ERROR*, and no additional OM attributes.

The OM attribute DS_PROBLEM, which is inherited from the superclass *DS_C_ERROR*, identifies the cause of this failure. Its value is one of the following:

- DS_E_INAPPROP_AUTHENTICATION
- DS_E_INSUFFICIENT_ACCESS_RIGHTS
- DS_E_INVALID_CREDENTIALS
- DS_E_INVALID_SIGNATURE
- DS_E_NO_INFO
- DS_E_PROTECTION_REQUIRED

DS_C_SERVICE_ERROR

An instance of OM class DS_C_SERVICE_ERROR reports a directory service error related to the provision of the service.

An application is not permitted to create or modify instances of this OM class. An instance of this OM class has the OM attributes of its superclasses, *OM_C_OBJECT* and *DS_C_ERROR*, and no additional OM attributes.

The OM attribute DS_PROBLEM, which is inherited from the superclass *DS_C_ERROR*, identifies the cause of the failure. Its value is one of the following:

- DS_E_ADMIN_LIMIT_EXCEEDED
- DS_E_BUSY
- DS_E_CHAINING_REQUIRED
- DS_E_DIT_ERROR
- DS_E_INVALID_REF
- DS_E_LOOP_DETECTED
- DS_E_OUT_OF_SCOPE
- DS_E_TIME_LIMIT_EXCEEDED
- DS_E_UNABLE_TO_PROCEED
- DS_E_UNAVAILABLE
- DS_E_UNAVAILABLE_CRIT_EXT
- DS_E_UNWILLING_TO_PERFORM

DS_C_SESSION

An instance of OM class DS_C_SESSION identifies a particular link from the application program to a DUA.

An instance of this OM class has the OM attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attributes listed in Table 46.

Table 46. OM Attributes of DS_C_SESSION

OM Attribute	**Value Syntax**	**Value Length**	**Value Number**	**Value Initially**
DS_DSA_ADDRESS	Object *(DS_C_ ADDRESS)*	—	0 or 1	*local*[2]
DS_DSA_NAME	Object *(DS_C_ NAME)*	—	0 or 1	*local*[1]
DS_FILE_ DESCRIPTOR	OM_S_INTEGER	—	1	See text
DS_ REQUESTOR	Object *(DS_C_ NAME)*	—	0 or 1	**NULL**

1 The default values of these OM attributes are set to the address and name of the default DSA entry in the local cache.

2 If this cache entry is not present, then these OM attributes are absent.

The DS_C_SESSION gathers all the information that describes a particular directory interaction. The parameters that will control such a session are set up in an instance of this OM class, which is then passed as an argument to **ds_bind( )**. This sets the OM attributes that describe the actual characteristics of this session, and then starts the session. A session started in this way must pass as the first argument to each interface function. The result of modifying an initiated session is unspecified. Finally, **ds_unbind( )** is used to terminate the session, after which the parameters can be modified and a new session started using the same instance, if required. Multiple concurrent sessions can run using multiple instances of this OM class.

The OM attributes of a session are as follows:

- DS_DSA_ADDRESS

  This attribute indicates the address of the default DSA named by DS_DSA_NAME.
- DS_DSA_NAME

  This attribute indicates the DN of the DSA that is used by default to service directory requests.
- DS_FILE_DESCRIPTOR (Optional Functionality)

  This OM attribute is not used by DCE XDS and is always set to DS_NO_VALID_FILE_DESCRIPTOR.
- DS_REQUESTOR

  This attribute is the DN of the user of this directory service session.

Applications can assume that an object of OM class DS_C_SESSION, created with default values of all its OM attributes, works with all the interface functions. Local administrators need to ensure that this is the case. Such a session can be created by passing the constant DS_DEFAULT_SESSION as an argument to **ds_bind( )**.

DS_C_SYSTEM_ERROR

An instance of OM class DS_C_SYSTEM_ERROR reports an error that occurred in the underlying operating system.

An application is not permitted to create or modify instances of this OM class. An instance of this OM class has the OM attributes of its superclasses, *OM_C_OBJECT* and *DS_C_ERROR*, and no additional OM attributes, although there can be additional implementation-defined OM attributes.

The OM attribute DS_PROBLEM, which is inherited from the superclass *DS_C_ERROR*, identifies the cause of the failure. Its value is the same as that of errno defined in the C language.

If such an error persists, a DS_C_LIBRARY_ERROR (DS_E_MISCELLANEOUS) is reported.

DS_C_UPDATE_ERROR

An instance of OM class DS_C_UPDATE_ERROR reports a directory service error peculiar to a modification operation.

An application is not permitted to create or modify instances of this OM class. An instance of this OM class has the OM attributes of its superclasses, *OM_C_OBJECT* and *DS_C_ERROR*, and no additional OM attributes.

The OM attribute DS_PROBLEM, which is inherited from the superclass *DS_C_ERROR*, identifies the cause of the failure. Its value is one of the following:

- DS_E_AFFECTS_MULTIPLE_DSAS
- DS_E_ENTRY_EXISTS
- DS_E_NAMING_VIOLATION
- DS_E_NOT_ALLOWED_ON_NON_LEAF
- DS_E_NOT_ALLOWED_ON_RDN
- DS_E_OBJECT_CLASS_MOD_PROHIB
- DS_E_OBJECT_CLASS_VIOLATION

Chapter 11. Basic Directory Contents Package

The standards define a number of attribute types (known as the *selected attribute types*), attribute syntaxes, attribute sets, and object classes (known as the *selected object classes* [1]). These definitions allow the creation and maintenance of directory entries for a number of common objects so that the representation of all such objects is the same throughout the directory. They include such objects as `Country`, `Person`, and `Organization`.

This chapter outlines names for each of these items, and defines OM classes to represent those that are not represented directly by OM syntaxes. The attribute values in the directory are not restricted to those discussed in this chapter, and new attribute types and syntaxes can be created at any time. (For further information on how the values of other syntaxes are represented in the interface, see "Chapter 9. XDS Interface Description" on page 181.)

The constants and OM classes in this chapter are defined in addition to those in "Chapter 10. XDS Class Definitions" on page 189, since they are not essential to the working of the interface, but instead allow directory entries to be utilized. The definitions belong to the basic directory contents package (BDCP), which is supported by the DCE XDS API following negotiation of its use with **ds_version( )**.

The object identifier associated with the BDCP is

```
{iso(1) identified-organization(3) icd-ecma(0012) member-company(2)
dec(1011) xopen(28) bdcp(1)}
```

It takes the following encoding:

```
\x2B\xC\x2\x87\x73\x1C\x1
```

This identifier is represented by the constant `DS_BASIC_DIR_CONTENTS_PKG`. The C constants associated with this package are in the **xdsbdcp.h** header file. (See the *IBM DCE Version 3.1 for AIX and Solaris: Application Development Reference*.)

The concepts and notation used are introduced in "Chapter 10. XDS Class Definitions" on page 189. A complete explanation of the meaning of the attributes and object classes is not given since this is beyond the scope of this guide. The purpose here is simply to present the representation of these items in the interface.

The selected attribute types are presented first, followed by the selected object classes. Next, the OM class hierarchy and OM class definitions required to support the selected attribute types are presented.

Note: This package should only be needed if a cell name is defined using the X.500 format (for example, **C=US/O=Acme Pepper Co/OU=Research**). As in "Chapter 10. XDS Class Definitions" on page 189, all definitions for this package are provided for completeness; however, only those OM objects required for an X.500 cell name definition should be needed for XDS/XOM over CDS.

1. These definitions are chiefly in *The Directory: Selected Attribute Types* (ISO 9594-6, CCITT X.520) and *The Directory: Selected Object Classes* (ISO 9594-7, CCITT X.521) with additional material in *The Directory: Overview of Concepts, Models, and Services* (ISO 9594-1, CCITT X.500) and *The Directory: Authentication Framework* (ISO 9594-8, CCITT X.509).

Selected Attribute Types

This section presents the attribute types, defined in the standards, which are to be used in directory entries. Each directory entry is composed of a number of attributes, each of which comprises an attribute type together with one or more attribute values. The form of each value of an attribute is determined by the attribute syntax associated with the attribute's type.

In the interface, attributes are displayed as instances of OM class `DS_C_ATTRIBUTE` with the attribute type represented as the value of the OM attribute `DS_ATTRIBUTE_TYPE`, and the attribute value (or values) represented as the value (or values) of the OM attribute `DS_ATTRIBUTE_VALUES`. Each attribute type has an object identifier, assigned in the standards, which is the value of the OM attribute `DS_ATTRIBUTE_TYPE`. These object identifiers are represented in the interface by constants with the same name as the directory attribute, and they are prefixed with `DS_A_` so that they can be easily identified.

Table 47 shows the names of the attribute types defined in the standards, together with the BER encoding of the object identifiers associated with each of them. Table 48 on page 221 shows the names of the attribute types, together with the OM value syntax that is used in the interface to represent values of that attribute type. Table 48 on page 221 also includes the range of lengths permitted for the string types. This indicates whether the attribute can be multivalued and which matching rules are provided for the syntax. Following the table is a brief description of each attribute.

The standards define matching rules that are used for deciding whether two values are equal (E), for ordering (O) two values, and for identifying one value as a substring (S) of another in directory service operations. Specific matching rules are given in this chapter for certain attributes. In addition, the following general rules apply as indicated:

- All attribute values whose syntax is String(`OM_S_NUMERIC_STRING`), String(`OM_S_PRINTABLE_STRING`), or String(`OM_S_TELETEX_STRING`) are considered insignificant for the following reasons:
  - Differences caused by the presence of spaces preceding the first printing character
  - Spaces following the last printing character
  - More than one consecutive space anywhere within the value
- For all attribute values whose syntax is String(`OM_S_TELETEX_STRING`), differences in the case of alphabetical characters are considered insignificant.

Note: The third and fourth columns of Table 47 contain the contents octets of the BER encoding of the object identifier. All these object identifiers stem from the root `{joint-iso-ccitt(2) ds(5) attributeType(4)}`.

Table 47. Object Identifiers for Selected Attribute Types

		Object Identifier BER	
Package	**Attribute Type**	**Decimal**	**Hexadecimal**
BDCP	`DS_A_ALIASED_OBJECT_NAME`	85, 4, 1	\x55\x04\x01
BDCP	`DS_A_BUSINESS_CATEGORY`	85, 4, 15	\x55\x04\x0F
BDCP	`DS_A_COMMON_NAME`	85, 4, 3	\x55\x04\x03
BDCP	`DS_A_COUNTRY_NAME`	85, 4, 6	\x55\x04\x06

Table 47. Object Identifiers for Selected Attribute Types (continued)

		Object Identifier BER	
Package	Attribute Type	Decimal	Hexadecimal
BDCP	DS_A_DESCRIPTION	85, 4, 13	\x55\x04\x0D
BDCP	DS_A_DEST_INDICATOR	85, 4, 27	\x55\x04\x1B
BDCP	DS_A_FACSIMILE_PHONE_NBR	85, 4, 23	\x55\x04\x17
BDCP	DS_A_INTERNAT_ISDN_NBR	85, 4, 25	\x55\x04\x19
BDCP	DS_A_KNOWLEDGE_INFO	85, 4, 2	\x55\x04\x02
BDCP	DS_A_LOCALITY_NAME	85, 4, 7	\x55\x04\x07
BDCP	DS_A_MEMBER	85, 4, 31	\x55\x04\x1F
BDCP	DS_A_OBJECT_CLASS	85, 4, 0	\x55\x04\x00
BDCP	DS_A_ORG_NAME	85, 4, 10	\x55\x04\x0A
BDCP	DS_A_ORG_UNIT_NAME	85, 4, 11	\x55\x04\x0B
BDCP	DS_A_OWNER	85, 4, 32	\x55\x04\x20
BDCP	DS_A_PHYS_DELIV_OFF_NAME	85, 4, 19	\x55\x04\x13
BDCP	DS_A_POST_OFFICE_BOX	85, 4, 18	\x55\x04\x12
BDCP	DS_A_POSTAL_ADDRESS	85, 4, 16	\x55\x04\x10
BDCP	DS_A_POSTAL_CODE	85, 4, 17	\x55\x04\x11
BDCP	DS_A_PREF_DELIV_METHOD	85, 4, 28	\x55\x04\x1C
BDCP	DS_A_PRESENTATION_ADDRESS	85, 4, 29	\x55\x04\x1D
BDCP	DS_A_REGISTERED_ADDRESS	85, 4, 26	\x55\x04\x1A
BDCP	DS_A_ROLE_OCCUPANT	85, 4, 33	\x55\x04\x21
BDCP	DS_A_SEARCH_GUIDE	85, 4, 14	\x55\x04\x0E
BDCP	DS_A_SEE_ALSO	85, 4, 34	\x55\x04\x22
BDCP	DS_A_SERIAL_NBR	85, 4, 5	\x55\x04\x05
BDCP	DS_A_STATE_OR_PROV_NAME	85, 4, 8	\x55\x04\x08
BDCP	DS_A_STREET_ADDRESS	85, 4, 9	\x55\x04\x09
BDCP	DS_A_SUPPORT_APPLIC_CONTEXT	85, 4, 3	\x55\x04\x03
BDCP	DS_A_SURNAME	85, 4, 4	\x55\x04\x04
BDCP	DS_A_PHONE_NBR	85, 4, 20	\x55\x04\x14
BDCP	DS_A_TELETEX_TERM_IDENT	85, 4, 22	\x55\x04\x16
BDCP	DS_A_TELEX_NBR	85, 4, 21	\x55\x04\x15
BDCP	DS_A_TITLE	85, 4, 12	\x55\x04\x0C
BDCP	DS_A_USER_PASSWORD	85, 4, 35	\x55\x04\x23
BDCP	DS_A_X121_ADDRESS	85, 4, 24	\x55\x04\x18

Table 48. Representation of Values for Selected Attribute Types

Attribute Type	OM Value Syntax	Value Length	Multi-valued	Matching Rules
DS_A_ALIASED_ OBJECT_NAME	Object *(DS_C_ NAME)*	—	no	E
DS_A_BUSINESS_ CATEGORY	String(OM_S_TELETEX_ STRING)	1–128	yes	E, S

Table 48. Representation of Values for Selected Attribute Types (continued)

Attribute Type	OM Value Syntax	Value Length	Multi-valued	Matching Rules
DS_A_COMMON_NAME	String(OM_S_TELETEX_STRING)	1–64	yes	E, S
DS_A_COUNTRY_NAME	String(OM_S_PRINTABLE_STRING)[1]	2	no	E
DS_A_DESCRIPTION	String(OM_S_TELETEX_STRING)	1–1024	yes	E, S
DS_A_DEST_ INDICATOR	String(OM_S_PRINTABLE_STRING)[2]	1–128	yes	E, S
DS_A_FACSIMILE_PHONE_NBR	Object(DS_C_ FACSIMILE_PHONE_NBR)	—	yes	—
DS_A_INTERNAT_ ISDN_NBR	String(OM_S_NUMERIC_STRING)[3]	1–16	yes	—
DS_A_ KNOWLEDGE_INFO	String(OM_S_TELETEX_STRING)	—	yes	E, S
DS_A_LOCALITY_ NAME	String(OM_S_TELETEX_STRING)	1–128	yes	E, S
DS_A_MEMBER	Object *(DS_C_ NAME)*	—	yes	E
DS_A_OBJECT_CLASS	String(OM_S_OBJECT_IDENTIFIER_STRING)	—	yes	E
DS_A_ORG_NAME	String(OM_S_TELETEX_STRING)	1–64	yes	E, S
DS_A_ORG_UNIT_NAME	String(OM_S_TELETEX_STRING)	1–64	yes	E, S
DS_A_OWNER	Object *(DS_C_NAME)*	—	yes	E
DS_A_PHYS_DELIV_OFF_NAME	String(OM_S_TELETEX_STRING)	1–128	yes	E, S
DS_A_POST_ OFFICE_BOX	String(OM_S_TELETEX_STRING)	1–40	yes	E, S
DS_A_POSTAL_ADDRESS	Object(DS_C_ POSTAL_ADDRESS)	—	yes	E
DS_A_POSTAL_CODE	String(OM_S_TELETEX_STRING)	1–40	yes	E, S
DS_A_PREF_ DELIV_METHOD	Enum(DS_Preferred_Delivery_Method)	—	yes	—
DS_A_PRESENTATION_ADDRESS	Object(DS_C_PRESENTATION_ ADDRESS)	—	no	E
DS_A_REGISTERED_ADDRESS	Object(DS_C_POSTAL_ADDRESS)	—	yes	—
DS_A_ROLE_OCCUPANT	Object *(DS_C_ NAME)*	—	yes	E
DS_A_SEARCH_GUIDE	Object(DS_C_SEARCH_GUIDE)	—	yes	—
DS_A_SEE_ALSO	Object *(DS_C_NAME)*	—	yes	E
DS_A_SERIAL_NBR	String(OM_S_PRINTABLE_STRING)	1–64	yes	E, S
DS_A_STATE_OR_PROV_NAME	String(OM_S_TELETEX_STRING)	1–128	yes	E, S

Table 48. Representation of Values for Selected Attribute Types (continued)

Attribute Type	OM Value Syntax	Value Length	Multi-valued	Matching Rules
DS_A_STREET_ ADDRESS	String(OM_S_OBJECT_ IDENTIFIER_STRING)	1–128	yes	E, S
DS_A_SUPPORT_ APPLIC_CONTEXT	String(OM_S_OBJECT_ IDENTIFIER_STRING)	—	yes	E
DS_A_SURNAME	String(OM_S_ TELETEX_STRING)	1–64	yes	E, S
DS_A_PHONE_NBR	String(OM_S_ PRINTABLE_STRING)[4]	1–32	yes	E, S
DS_A_TELETEX_ TERM_IDENT	Object(DS_C_TELETEX_ TERM_IDENT)	—	yes	—
DS_A_TELEX_NBR	Object(DS_C_TELEX_ NBR)	—	yes	—
DS_A_TITLE	String(OM_S_TELETEX_ STRING)	1–64	yes	E, S
DS_A_USER_PASSWORD	String(OM_S_OCTET_ STRING)	0–128	yes	—
DS_A_X121_ADDRESS	String(OM_S_NUMERIC_ STRING)[5]	1–15	yes	E, S

[1] As permitted by ISO 3166.

[2] As permitted by Recommendations F.1 and F.31.

[3] As permitted by E.164.

[4] As permitted by E.123 (for example, +44 582 10101).

[5] As permitted by X.121.

Throughout the descriptions that follow, the term *object* indicates the directory object whose directory entry contains the corresponding directory attributes.

- DS_A_ALIASED_OBJECT_NAME

  This attribute occurs only in alias entries. It assigns the distinguished name (DN) of the object, provided with an alias, using the entry in which this attribute occurs. An alias is an alternative to an object's DN. Any object can (but need not) have one or more aliases. The directory service is said to dereference an alias whenever it replaces the alias during name processing with the DN associated with it by means of this attribute.
- DS_A_BUSINESS_CATEGORY

  This attribute provides descriptions of the businesses in which the object is engaged.
- DS_A_COMMON_NAME

  This attribute provides the names by which the object is commonly known in the context defined by its position in the DIT. The names can conform to the naming convention of the country or culture with which the object is associated. They can be ambiguous.
- DS_A_COUNTRY_NAME

  This attribute identifies the country in which the object is located or with which it is associated in some other important way. The matching rules require that differences in the case of alphabetical characters be considered insignificant. It has a length of two characters and its values are those listed in ISO 3166.
- DS_A_DESCRIPTION

This attribute gives informative descriptions of the object.

- DS_A_DEST_INDICATOR

  This attribute provides the country-city pairs by means of which the object can be reached via the public telegram service. The matching rules require that differences in the case of alphabetical characters be considered insignificant.
- DS_A_FACSIMILE_PHONE_NBR

  This attribute provides the telephone numbers for facsimile terminals (and their parameters, if required) by means of which the object can be reached or with which it is associated in some other important way.
- DS_A_INTERNAT_ISDN_NBR

  This attribute provides the international ISDN numbers by means of which the object can be reached or with which it is associated in some other important way. The matching rules require that differences caused by the presence of spaces be considered insignificant.
- DS_A_KNOWLEDGE_INFO

  This attribute occurs only in entries that describe a DSA. It provides a human-intelligible accumulated description of the directory knowledge possessed by the DSA.
- DS_A_LOCALITY_NAME

  This attribute identifies geographical areas or localities. When used as part of a directory name, it specifies the localities in which the object is located or with which it is associated in some other important way.
- DS_A_MEMBER

  This attribute gives the names of objects that are considered members of the present object; for example, a distribution list for electronic mail.
- DS_A_OBJECT_CLASS

  This attribute identifies the object classes to which the object belongs, and it also identifies their superclasses. All such object classes that have object identifiers assigned to them are present, except that object class DS_O_TOP need not (but can) be present provided that some other value is present. This attribute must be present in every entry and cannot be modified. For a further discussion, see “Selected Object Classes” on page 226.
- DS_A_ORG_NAME

  This attribute identifies organizations. When used as part of a directory name, it specifies an organization with which the object is affiliated. Several values can identify the same organization in different ways.
- DS_A_ORG_UNIT_NAME

  This attribute identifies organizational units. When used as part of a directory name, it specifies an organizational unit with which the object is affiliated. The units are understood to be parts of the organization that the DS_A_ORG_NAME attribute indicates. Several values can identify the same unit in different ways.
- DS_A_OWNER

  This attribute gives the names of objects that have responsibility for the object.
- DS_A_PHYS_DELIV_OFF_NAME

  This attribute gives the names of cities, towns, villages, and so on, that contain physical delivery offices through which the object can take delivery of physical mail.
- DS_A_POST_OFFICE_BOX

This attribute identifies post office boxes at which the object can take delivery of physical mail. This information is also displayed as part of the DS_A_POSTAL_ADDRESS attribute, if it is present.

- DS_A_POSTAL_ADDRESS

  This attribute gives the postal addresses at which the object can take delivery of physical mail. The matching rules require that differences in the case of alphabetical characters be considered insignificant.
- DS_A_POSTAL_CODE

  This attribute gives the postal codes that are assigned to areas or buildings through which the object can take delivery of physical mail. This information is also displayed as part of the DS_A_POSTAL_ADDRESS attribute, if it is present.
- DS_A_PREF_DELIV_METHOD

  This attribute gives the object's preferred methods of communication, in the order of preference. The values are as follows:
  - DS_ANY_DELIV_METHOD, meaning that the object has no preference.
  - DS_G3_FACSIMILE_DELIV, meaning via the Group 3 facsimile.
  - DS_G4_FACSIMILE_DELIV, meaning via the Group 4 facsimile.
  - DS_IA5_TERMINAL_DELIV, meaning via the IA5 text.
  - DS_MHS_DELIV, meaning via X.400.
  - DS_PHYS_DELIV, meaning via the postal or other physical delivery system.
  - DS_PHONE_DELIV, meaning via telephone.
  - DS_TELETEX_DELIV, meaning via teletex.
  - DS_TELEX_DELIV, meaning via telex.
  - DS_VIDEOTEX_DELIV, meaning via videotex.
- DS_A_PRESENTATION_ADDRESS

  This attribute contains the OSI presentation address of the object, which is an OSI application entity. The matching rule for a presented value to match a value stored in the directory is that the P-Selector, S-Selector, and T-Selector of the two presentation addresses must be equal, and the N-Addresses of the presented value must be a subset of those of the stored value.
- DS_A_REGISTERED_ADDRESS

  This attribute contains mnemonics by means of which the object can be reached via the public telegram service, according to Recommendation F.1. A mnemonic identifies an object in the context of a particular city, and it is registered in the country containing the city. The matching rules require that differences in the case of alphabetical characters be considered insignificant.
- DS_A_ROLE_OCCUPANT

  This attribute occurs only in entries that describe an organizational role. It gives the names of objects that fulfill the organizational role.
- DS_A_SEARCH_GUIDE

  This attribute contains the criteria that can be used to build filters for conducting searches in which the object is the base object.
- DS_A_SEE_ALSO

  This attribute contains the names of objects that represent other aspects of the real-world object that the present object represents.
- DS_A_SERIAL_NBR

  This attribute contains the serial numbers of a device.
- DS_A_STATE_OR_PROV_NAME

This attribute specifies a state or province. When used as part of a directory name, it identifies states, provinces, or other geographical regions in which the object is located or with which it is associated in some other important way.

- `DS_A_STREET_ADDRESS`

  This attribute identifies a site for the local distribution and physical delivery of mail. When used as part of a directory name, it identifies the street address (for example, street name and house number) at which the object is located or with which it is associated in some other important way.
- `DS_A_SUPPORT_APPLIC_CONTEXT`

  This attribute occurs only in entries that describe an OSI application entity. It identifies OSI application contexts supported by the object.
- `DS_A_SURNAME`

  This attribute occurs only in entries that describe individuals. The surname by which the individual is commonly known, normally inherited from the individual's parent (or parents) or taken at marriage, as determined by the custom of the country or culture with which the individual is associated.
- `DS_A_PHONE_NBR`

  This attribute identifies telephones by means of which the object can be reached or with which it is associated in some other important way. The matching rules require that differences caused by the presence of spaces and dashes be considered insignificant.
- `DS_A_TELETEX_TERM_IDENT`

  This attribute contains descriptions of teletex terminals by means of which the object can be reached or with which it is associated in some other important way.
- `DS_A_TELEX_NBR`

  This attribute contains descriptions of telex terminals by means of which the object can be reached or with which it is associated in some other important way.
- `DS_A_TITLE`

  This attribute identifies positions or functions of the object within its organization.
- `DS_A_USER_PASSWORD`

  This attribute contains the passwords assigned to the object.
- `DS_A_X121_ADDRESS`

  This attribute identifies points on the public data network at which the object can be reached or with which it is associated in some other important way. The matching rules require that differences caused by the presence of spaces be considered insignificant.

Selected Object Classes

This section presents the object classes that are defined in the standards. Object classes are groups of directory entries that share certain characteristics. The object classes are arranged into a lattice, based on the object class `DS_O_TOP`. In a lattice, each element, except a leaf, has one or more immediate subordinates but also has one or more immediate superiors. This contrasts with a tree, where each element has exactly one immediate superior. Object classes closer to `DS_O_TOP` are called *superclasses*, and those further away are called subclasses. This relationship is not connected to any other such relationship in this guide.

Each directory entry belongs to an object class, and to all the superclasses of that object class. Each entry has an attribute named `DS_A_OBJECT_CLASS`, which was discussed in the previous section, and which identifies the object classes to which the entry belongs. The values of this attribute are object identifiers, which are represented in the interface by constants with the same name as the object class, prefixed by DS_O_.

Associated with each object class are zero or more mandatory and zero or more optional attributes. Each directory entry must contain all the mandatory attributes and can (but need not) contain the optional attributes associated with the object class and its superclasses.

The object classes defined in the standards are shown in Table 49, together with their object identifiers.

Note: The third and fourth columns of Table 49 contain the contents octets of the BER encoding of the object identifier. All these object identifiers stem from the root {joint-iso-ccitt(2) ds(5) objectClass(6)}.

Table 49. Object Identifiers for Selected Object Classes

		Object Identifier BER	
Package	Attribute Type	Decimal	Hexadecimal
BDCP	DS_O_ALIAS	85, 6, 1	\x55\x06\x01
BDCP	DS_O_APPLIC_ENTITY	85, 6, 12	\x55\x06\x0C
BDCP	DS_O_APPLIC_PROCESS	85, 6, 11	\x55\x06\x0B
BDCP	DS_O_COUNTRY	85, 6, 2	\x55\x06\x02
BDCP	DS_O_DEVICE	85, 6, 14	\x55\x06\x0E
BDCP	DS_O_DSA	85, 6, 13	\x55\x06\x0D
BDCP	DS_O_GROUP_OF_NAMES	85, 6, 9	\x55\x06\x09
BDCP	DS_O_LOCALITY	85, 6, 3	\x55\x06\x03
BDCP	DS_O_ORG	85, 6, 4	\x55\x06\x04
BDCP	DS_O_ORG_PERSON	85, 6, 7	\x55\x06\x07
BDCP	DS_O_ORG_ROLE	85, 6, 8	\x55\x06\x08
BDCP	DS_O_ORG_UNIT	85, 6, 5	\x55\x06\x05
BDCP	DS_O_PERSON	85, 6, 6	\x55\x06\x06
BDCP	DS_O_RESIDENTIAL_PERSON	85, 6, 10	\x55\x06\x0A
BDCP	DS_O_TOP	85, 6, 0	\x55\x06\x00

OM Class Hierarchy

The remainder of this chapter defines the additional OM classes used to represent values of the selected attributes described in “Selected Attribute Types” on page 220. Some of the selected attributes are represented by OM classes that are used in the interface itself, and hence are defined in “Chapter 10. XDS Class Definitions” on page 189; for example, *DS_C_NAME*.

This section shows the hierarchical organization of the OM classes that are defined in the following sections, and it shows which OM classes inherit additional OM attributes from their OM superclasses. In the following list, subclassification is indicated by indentation, and the names of abstract OM classes are in italics. For example, DS_C_POSTAL_ADDRESS is an immediate subclass of the abstract OM class *OM_C_OBJECT*.

OM_C_OBJECT
- DS_C_FACSIMILE_PHONE_NBR
- DS_C_POSTAL_ADDRESS
- DS_C_SEARCH_CRITERION
- DS_C_SEARCH_GUIDE
- DS_C_TELETEX_TERM_IDENT
- DS_C_TELEX_NBR

None of the OM classes in the preceding list are encodable by using **om_encode( )** and **om_decode( )**.

DS_C_FACSIMILE_PHONE_NBR

An instance of OM class DS_C_FACSIMILE_PHONE_NBR identifies and describes a facsimile terminal, if required.

An instance of this OM class has the OM attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attributes listed in Table 50.

Table 50. OM Attributes of DS_C_FACSIMILE_PHONE_NBR

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_PARAMETERS	Object (MH_C_G3_FAX_ NBPS)[1]	—	0 or 1	—
DS_PHONE_NBR	String(OM_S_ PRINTABLE_STRING)[2]	1–32	1	—

[1] As defined in the X.400 API specifications.

[2] As permitted by E.123 (for example, +44 582 10101).

- DS_PARAMETERS

  If present, this attribute identifies the facsimile terminal's nonbasic capabilities.
- DS_PHONE_NBR

  This attribute contains a telephone number by means of which the facsimile terminal is accessed.

DS_C_POSTAL_ADDRESS

An instance of OM class DS_C_POSTAL_ADDRESS is a postal address.

An instance of this OM class has the OM attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attribute listed in Table 51.

Table 51. OM Attribute of DS_C_POSTAL_ADDRESS

OM Attribute	**Value Syntax**	**Value Length**	**Value Number**	**Value Initially**
DS_POSTAL_ ADDRESS	String(OM_S_ TELETEX_STRING)	1–30	1–6	—

- DS_POSTAL_ADDRESS

  Each value of this OM attribute is one line of the postal address. It typically includes a name, street address, city name, state or province name, postal code, and possibly a country name.

DS_C_SEARCH_CRITERION

An instance of OM class DS_C_SEARCH_CRITERION is a component of a DS_C_SEARCH_GUIDE OM object.

An instance of this OM class has the OM attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attributes listed in Table 52.

Table 52. OM Attributes of DS_C_SEARCH_CRITERION

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_ATTRIBUTE_TYPE	String(OM_S_OBJECT_ IDENTIFIER_STRING)	—	0 or 1	—
DS_CRITERIA	Object(DS_C_SEARCH_ CRITERION)	—	0 or more	—
DS_FILTER_ITEM_TYPE	Enum(DS_Filter_Item_ Type)	—	0 or 1	—
DS_FILTER_TYPE	Enum(DS_Filter_Type)	—	1	DS_ITEM

A DS_C_SEARCH_CRITERION suggests how to build part of a filter to be used when searching the directory. Its meaning depends on the value of its OM attribute DS_FILTER_TYPE. If the value is DS_ITEM, then the criteria suggests building an instance of OM class DS_C_FILTER_ITEM. If DS_FILTER_TYPE has any other value, it suggests building an instance of OM class DS_C_FILTER.

- DS_ATTRIBUTE_TYPE

  This attribute indicates the attribute type to be used in the suggested DS_C_FILTER_ITEM. This OM attribute is only present when the value of DS_FILTER_TYPE is DS_ITEM.
- DS_CRITERIA

  This attribute contains nested search criteria. This OM attribute is not present when the value of DS_FILTER_TYPE is DS_ITEM.
- DS_FILTER_ITEM_TYPE

  This attribute indicates the type of suggested filter item. Its value can be one of the following:
  - DS_APPROXIMATE_MATCH
  - DS_EQUALITY
  - DS_GREATER_OR_EQUAL
  - DS_LESS_OR_EQUAL
  - DS_SUBSTRINGS

  However, the filter item cannot have the value DS_PRESENT. This OM attribute is only present when the value of DS_FILTER_TYPE is DS_ITEM.
- DS_FILTER_TYPE

  This attribute indicates the type of suggested filter. The value DS_ITEM means that the suggested component is a filter item, not a filter. The other values suggest the corresponding type of filter. Its value is one of the following:
  - DS_AND
  - DS_ITEM
  - DS_NOT
  - DS_OR

DS_C_SEARCH_GUIDE

An instance of OM class DS_C_SEARCH_GUIDE suggests a criteria for searching the directory for particular entries. It can be used to build a DS_C_FILTER parameter for **ds_search( )** operations that are based on the object in whose entry the search guide occurs.

An instance of this OM class has the OM attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attributes listed in Table 53.

Table 53. OM Attributes of DS_C_SEARCH_GUIDE

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_OBJECT_CLASS	String(OM_S_OBJECT_ IDENTIFIER_STRING)	—	0 or 1	—
DS_CRITERIA	Object(DS_C_SEARCH_ CRITERION)	—	1	—

- DS_OBJECT_CLASS

  This attribute identifies the object class of the entries to which the search guide applies. If this OM attribute is absent, the search guide applies to objects of any class.
- DS_CRITERIA

  This attribute contains the suggested search criteria.

DS_C_TELETEX_TERM_IDENT

An instance of OM class DS_C_TELETEX_TERM_IDENT identifies and describes a teletex terminal.

An instance of this OM class has the OM attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attributes listed in Table 54.

Table 54. OM Attributes of DS_C_TELETEX_TERM_IDENT

OM Attribute	**Value Syntax**	**Value Length**	**Value Number**	**Value Initially**
DS_PARAMETERS	Object(MH_C_ TELETEX_NBPS)[1]	—	0 or 1	—
DS_TELETEX_TERM	String(OM_S_ PRINTABLE_STRING)[2]	1–1024	1	—

[1] As defined in the X.400 API specifications.

[2] As permitted by F.200.

- DS_PARAMETERS

  This attribute identifies the teletex terminal's nonbasic capabilities.
- DS_TELETEX_TERM

  This attribute identifies the teletex terminal.

DS_C_TELEX_NBR

An instance of OM class DS_C_TELEX_NBR identifies and describes a telex terminal.

An instance of this OM class has the OM attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attributes listed in Table 55.

Table 55. OM Attributes of DS_C_TELEX_NBR

OM Attribute	**Value Syntax**	**Value Length**	**Value Number**	**Value Initially**
DS_ANSWERBACK	String(OM_S_ PRINTABLE_STRING)	1–8	1	—
DS_COUNTRY_CODE	String(OM_S PRINTABLE_STRING)	1–4	1	—
DS_TELEX_NBR	String(OM_S_ PRINTABLE_STRING)	1–14	1	—

- DS_ANSWERBACK

  This attribute contains the code with which the telex terminal acknowledges calls placed to it.
- DS_COUNTRY_CODE

  This attribute contains the identifier of the country through which the telex terminal is accessed.
- DS_TELEX_NBR

  This attribute contains the number by means of which the telex terminal is addressed.

Chapter 12. Information Syntaxes

This chapter defines the syntaxes permitted for attribute values. The syntaxes are closely aligned with the types and type constructors of ASN.1. The `OM_value` data type specifies how a value of each syntax is represented in the C interface (see “Chapter 13. XOM Service Interface” on page 239).

Syntax Templates

The names of certain syntaxes are constructed from *syntax templates*. A syntax template is a lexical construct comprising a primary identifier followed by an * (asterisk) enclosed in parentheses, as follows:

```
identifier (*)
```

A syntax template encompasses a group of related syntaxes. Any member of the group, without distinction, is indicated by the primary identifier (*identifier*) alone. A particular member is indicated by the template with the asterisk replaced by one of a set of secondary identifiers associated with the template, as follows:

$identifier_1$
(
$identifier_2$)

Syntaxes

A variety of syntaxes are defined. Most are functionally equivalent to ASN.1 types, as documented in “Relationship to ASN.1 Simple Types” on page 235 through “Relationship to ASN.1 Type Constructors” on page 236.

The following syntaxes are defined:

- `OM_S_BOOLEAN`

  A value of this syntax is a Boolean; that is, it can be `OM_TRUE` or `OM_FALSE`.
- Enum(*)

  A value of any syntax encompassed by this syntax template is one of a set of values associated with the syntax. The only significant characteristic of the values is that they are distinct.

  The group of syntaxes encompassed by this template is open-ended. Zero or more members are added to the group by each package definition. The secondary identifiers that indicate the members are also assigned there.
- `OM_S_INTEGER`

  A value of this syntax is a positive or negative integer.
- `OM_S_NULL`

  The one value of this syntax is a valueless placeholder.
- Object(*)

  A value of any syntax encompassed by this syntax template is an object, which is any instance of a class associated with the syntax.

  The group of syntaxes encompassed by this template is open-ended. One member is added to the group by each class definition. The secondary identifier that indicates the member is the name of the class.

- String(*)

  A value of any syntax encompassed by this syntax template is a string (as defined in "Strings") whose form and meaning are associated with the syntax.

  The group of syntaxes encompassed by this template is closed. One syntax is defined for each ASN.1 string type. The secondary identifier that indicates the member is, in general, the first word of the type's name.

Strings

A *string* is an ordered sequence of zero or more bits, octets, or characters accompanied by the string's length.

The value *length* of a string is the number of bits in a *bit string*, octets in an *octet string*, or characters in a *character string*. Any constraints on the value length of a string are specified in the appropriate class definitions. The length is confined to the range 0 to 2^{32}.

Note: The length of a character string does not necessarily equal the number of characters it comprises because, for example, a single character can be represented by using several octets.

The elements of a string are numbered. The position of the first element is 0 (zero). The positions of successive elements are successive positive integers.

The syntaxes that form the string group are identified in Table 56, which gives the secondary identifier assigned to each such syntax.

Note: The identifiers in the first, second, and third columns of Table 56 indicate the syntaxes of bit, octet, and character strings, respectively. The String group comprises all syntaxes identified in the table.

Table 56. String Syntax Identifiers

Bit String Identifier	Octet String Identifier	Character String Identifier
OM_S_BIT_STRING	OM_S_ENCODING_ STRING[1]	OM_S_GENERAL_ STRING[2]
	OM_S_OBJECT_ IDENTIFIER_STRING[3]	OM_S_GENERALIZED_TIME_STRING[2]
	OM_S_OCTET_STRING	OM_S_GRAPHIC_STRING[2]
		OM_S_IA5_STRING[2]
		OM_S_NUMERIC_STRING[2]
		OM_S_OBJECT_ DESCRIPTOR_STRING[2]
		OM_S_ PRINTABLE_STRING[2]
		OM_S_TELETEX_STRING[2]
		OM_S_UTC_TIME_STRING[2]
		OM_S_VIDEOTEX_STRING[2]
		OM_S_VISIBLE_STRING[2]

1 The octets are those that BER permits for the contents octets of the encoding of a value of any ASN.1 type.

2 The characters are those permitted by ASN.1's type of the corresponding name. Values of these syntaxes are represented in their BER-encoded form. The octets by which they are represented are those that BER permits for the contents octets of a primitive encoding of a value of that type.

3 The octets are those that BER permits for the contents octets of the encoding of a value of ASN.1's object identifier type.

Representation of String Values

In the service interface, a string value is represented by a string data type. This is defined in "Strings" on page 234. The length of a string is the number of octets by which it is represented at the interface. It is confined to the range 0 to 2^{32}.

The length of a character does not need to be equal to the number of characters it comprises because, for example, a single character can be represented by using several octets.

It may be necessary to segment large string values when passing them across the interface. A segment is any zero or more contiguous octets of a string value. Segment boundaries are without semantic significance.

Relationship to ASN.1 Simple Types

As shown in Table 57, for every ASN.1 simple type except Real, there is an OM syntax that is functionally equivalent to it. The simple types are listed in the first column of the table; the corresponding syntaxes are listed in the second column.

Table 57. Syntax for ASN.1 Simple Types

Type	**Syntax**
Bit String	String(`OM_S_BIT_STRING`)
Boolean	`OM_S_BOOLEAN`
Integer	`OM_S_INTEGER`
Null	`OM_S_NULL`
Object Identifier	String(`OM_S_OBJECT_IDENTIFIER_STRING`)
Octet String	String(`OM_S_OCTET_STRING`)
Real	None[1]

1 A future edition of XOM may define a syntax corresponding to this type.

Relationship to ASN.1 Useful Types

As shown in Table 58, for every ASN.1 useful type, there is an OM syntax that is functionally equivalent to it. The useful types are listed in the first column of the table; the corresponding syntaxes are listed in the second column.

Table 58. Syntaxes for ASN.1 Useful Types

Type	**Syntax**
External	Object(`OM_C_EXTERNAL`)

Table 58. Syntaxes for ASN.1 Useful Types (continued)

Type	Syntax
Generalized Time	String(OM_S_GENERALISED_TIME_STRING)
Object Descriptor	String(OM_S_OBJECT_DESCRIPTOR_STRING)
Universal Time	String(OM_S_UTC_TIME_STRING)

Relationship to ASN.1 Character String Types

As shown in Table 59, for every ASN.1 character string type, there is an OM syntax that is functionally equivalent to it. The ASN.1 character string types are listed in the first column of the table; the corresponding syntax is listed in the second column.

Table 59. Syntaxes for ASN.1 Character String Types

Type	Syntax
General String	String(OM_S_GENERAL_STRING)
Graphic String	String(OM_S_GRAPHIC_STRING)
IA5 String	String(OM_S_IA5_STRING)
—	String(OM_S_LOCAL_STRING)
Numeric String	String(OM_S_NUMERIC_STRING)
Printable String	String(OM_S_PRINTABLE_STRING)
Teletex String	String(OM_S_TELETEX_STRING)
Videotex String	String(OM_S_VIDEOTEX_STRING)
Visible String	String(OM_S_VISIBLE_STRING)

Relationship to ASN.1 Type Constructors

As shown in Table 60, there are functionally equivalent OM syntaxes for some (but not all) ASN.1 type constructors. The constructors are listed in the first column; corresponding syntaxes are listed in the second column.

Table 60. Syntaxes for ASN.1 Type Constructors

Type Constructor	Syntax
Any	String(OM_S_ENCODING_STRING)
Choice	OM_S_OBJECT
Enumerated	OM_S_ENUMERATION
Selection	None[1]
Sequence	OM_S_OBJECT
Sequence Of	OM_S_OBJECT
Set	OM_S_OBJECT
Set Of	OM_S_OBJECT
Tagged	None[2]

1 This type constructor, a purely specification-time phenomenon, has no corresponding syntax.

[2] This type constructor is used to distinguish the alternatives of a choice or the elements of a sequence or set, a function performed by attribute types.

The effects of the principal type constructors can be achieved, in any of a variety of ways, by using objects-to-group attributes or using attributes-to-group values. An OM application designer can (but need not) model these constructors as classes of the following kinds:

- Choice

  An attribute type can be defined for each alternative, with just one being permitted in an instance of the class.
- Sequence or Set

  An attribute type can be defined for each sequence or set element. If an element is optional, then the attribute has zero or one value.
- Sequence Of or Set Of

  A single multivalued attribute can be defined.

An ASN.1 definition of an enumerated type component of a structured type is generally mapped to an OM attribute with an OM syntax `OM_S_ENUMERATION` in this interface. Where the ASN.1 component is optional, this is generally indicated by an additional member of the enumeration, rather than by the omission of the OM attribute. This leads to simpler programming in the application.

Chapter 13. XOM Service Interface

This chapter describes the following aspects of the XOM service interface:

- The conformance of the DCE X/Open OSI-Abstract-Data Manipulation (XOM) implementation to the X/Open specification.
- The data types whose data values are the parameters and results of the functions that the service makes available to the client.
- An overview of the functions that the service makes available to the client. For a complete description of these functions, see the corresponding reference pages in *IBM DCE Version 3.1 for AIX and Solaris: Application Development Reference*.
- The return codes that indicate the outcomes (in particular, the exceptions) that the functions can report.

See the *IBM DCE Version 3.1 for AIX and Solaris: Application Development Reference* for examples of using the XOM interface.

Standards Conformance

The DCE XOM implementation conforms to the following specification:

X/Open CAE Specification, *OSI-Abstract-Data Manipulation (XOM)* (November 1991)

The following apply to the DCE XOM implementation:

- Multiple workspaces for XDS objects are supported.
- The OM package is supported.
- The **om_encode( )** and **om_decode( )** functions are not supported. The transfer of objects between workspaces is not envisaged within the DCE environment. The OM classes used by the DCE XDS/XOM API are not encodable.
- Translation to local character sets is not supported.

XOM Data Types

The data types of the XOM service interface are defined in this section and listed in Table 61. These data types are repeated in the XOM reference pages (see **xom.h(4xom)**).

Table 61. XOM Service Interface Data Types

Data Type	**Description**
OM_boolean	Type definition for a Boolean data value.
OM_descriptor	Type definition for describing an attribute type and value.
OM_enumeration	Type definition for an Enumerated data value.
OM_exclusions	Type definition for the *exclusions* parameter for **om_get( )**.
OM_integer	Type definition for an Integer data value.
OM_modification	Type definition for the *modification* parameter for **om_put( )**.

Table 61. XOM Service Interface Data Types (continued)

Data Type	Description
OM_object	Type definition for a handle to either a private or a public object.
OM_object_identifier	Type definition for an object identifier data value.
OM_private_object	Type definition for a handle to an object in an implementation-defined, or private, representation.
OM_public_object	Type definition for a defined representation of an object that can be directly interrogated by a programmer.
OM_return_code	Type definition for a value returned from all OM functions, indicating either that the function succeeded or why it failed.
OM_string	Type definition for a data value of one of the String syntaxes.
OM_syntax	Type definition for identifying a syntax type.
OM_type	Type definition for identifying an OM attribute type.
OM_type_list	Type definition for enumerating a sequence of OM attribute types.
OM_value	Type definition for representing any data value.
OM_value_length	Type definition for indicating the number of bits, octets, or characters in a string.
OM_value_position	Type definition for designating a particular location within a String data value.
OM_workspace	Type definition for identifying an application-specific API that implements OM, such as directory or message handling.

Some data types are defined in terms of the following *intermediate data types*, whose precise definitions in C are defined by the system:

- `OM_sint`

  The positive and negative integers that can be represented in 16 bits
- `OM_sint16`

  The positive and negative integers that can be represented in 16 bits
- `OM_sint32`

  The positive and negative integers that can be represented in 32 bits
- `OM_uint`

  The nonnegative integers that can be represented in 16 bits
- `OM_uint16`

  The nonnegative integers that can be represented in 16 bits
- `OM_uint32`

  The nonnegative integers that can be represented in 32 bits

Note: The `OM_sint` and `OM_uint` data types are defined by the range of integers they must accommodate. As typically declared in the C interface, they are defined by the range of integers permitted by the host machine's word size. The latter range, however, always encompasses the former.

The type definitions for these data types are as follows:

```
typedef int             OM_sint;
typedef short           OM_sint16;
typedef long int        OM_sint32;
typedef unsigned        OM_uint;
typedef unsigned short  OM_uint16;
typedef long unsigned   OM_uint32;
```

OM_boolean

The C declaration for an `OM_boolean` data value is as follows:

```
typedef OM_uint32 OM_boolean;
```

A data value of this data type is a Boolean; that is, either FALSE or TRUE.

FALSE (`OM_FALSE`) is indicated by 0 (zero). TRUE is indicated by any other integer, although the symbolic constant `OM_TRUE` refers to the integer 1 specifically.

OM_descriptor

The `OM_descriptor` data type is used to describe an attribute type and value. Its C declaration is as follows:

```
typedef struct OM_descriptor_struct
{
   OM_type        type;
   OM_syntax      syntax;
   union  OM_value_union value;
}  OM_descriptor;
```

Note: Other components are encoded in high bits of the syntax member.

See the `OM_value` data type described in "OM_value" on page 249 or the **xom.h(4xom)** reference page for a description of the `OM_value_union` structure.

A data value of this type is a descriptor, which embodies an attribute value. An array of descriptors can represent all the values of all the attributes of an object, and is the representation called `OM_public_object`. A descriptor has the following components:

- *type*

  An `OM_type` data type. It identifies the data type of the attribute value.
- *syntax*

  An `OM_syntax` data type. It identifies the syntax of the attribute value. Components 3 to 7 (that is, the components *long-string* through *private* that follow) are encoded in the high-order bits of this structure member. Therefore, the syntax always needs to be masked with the constant `OM_S_SYNTAX`. An example is the following:

  ```
  my_syntax = my_public_object[3].syntax &
              OM_S_SYNTAX;

  my_public_object[4].syntax =
  my_syntax + (my_public_object[4].syntax &
  ~OM_S_SYNTAX);
  ```
- *long-string*

An OM_boolean data type. It is OM_TRUE only if the descriptor is a service-generated descriptor and the length of the value is greater than an implementation-defined limit.

This component occupies bit 15 (0x8000) of the syntax and is represented by the constant OM_S_LONG_STRING.

- *no-value*

  An OM_boolean data type. It is OM_TRUE only if the descriptor is a service-generated descriptor and the value is not present because OM_EXCLUDE_VALUES or OM_EXCLUDE_MULTIPLES is set in om_get( ).

  This component occupies bit 14 (0x4000) of the syntax and is represented by the constant OM_S_NO_VALUE.

- *local-string*

  An OM_boolean data type, significant only if the syntax is one of the string syntaxes. It is OM_TRUE only if the string is represented in an implementation-defined local character set. The local character set may be more amenable for use as keyboard input or display output than the nonlocal character set, and it can include specific treatment of line termination sequences. Certain interface functions can convert information in string syntaxes to or from the local representation, which may result in a loss of information.

  This component occupies bit 13 (0x2000) of the syntax and is represented by the constant OM_S_LOCAL_STRING. The DCE XOM implementation does not support translation of strings to a local character set.

- *service-generated*

  An OM_boolean data type. It is OM_TRUE only if the descriptor is a service-generated descriptor and the first descriptor of a public object, or the defined part of a private object (see the *(3xom) reference pages).

  This component occupies bit 12 (0x1000) of the syntax and is represented by the constant OM_S_SERVICE_GENERATED.

- *private*

  An OM_boolean data type. It is OM_TRUE only if the descriptor in the service-generated public object contains a reference to the handle of a private subobject, or in the defined part of a private object.

  Note: This applies only when the descriptor is a service-generated descriptor. The client need not set this bit in a client-generated descriptor that contains a reference to a private object.

  In the C interface, this component occupies bit 11 (0x0800) of the syntax and is represented by the constant OM_S_PRIVATE.

- *value*

  An OM_value data type. It identifies the attribute value.

OM_enumeration

The OM_enumeration data type is used to indicate an Enumerated data value. Its C declaration is as follows:

```
typedef OM_sint32 OM_enumeration;
```

A data value of this data type is an attribute value whose syntax is OM_S_ENUMERATION.

OM_exclusions

The `OM_exclusions` data type is used for the *exclusions* parameter of om_get( ). Its C declaration is as follows:

```
typedef OM_uint OM_exclusions;
```

A data value of this data type is an unordered set of one or more values, all of which are distinct. Each value indicates an exclusion, as defined by **om_get( )**, and is chosen from the following set:

- `OM_EXCLUDE_ALL_BUT_THESE_TYPES`
- `OM_EXCLUDE_MULTIPLES`
- `OM_EXCLUDE_ALL_BUT_THESE_VALUES`
- `OM_EXCLUDE_VALUES`
- `OM_EXCLUDE_SUBOBJECTS`
- `OM_EXCLUDE_DESCRIPTORS`

Alternatively, the single value `OM_NO_EXCLUSIONS` can be chosen; this selects the entire object.

Each value except `OM_NO_EXCLUSIONS` is represented by a distinct bit. The presence of the value is represented as 1; its absence is represented as 0 (zero). Thus, multiple exclusions are requested by ORing the values that indicate the individual exclusions.

OM_integer

The `OM_integer` data type is used to indicate an integer data value. Its C declaration is as follows:

```
typedef OM_sint32 OM_integer;
```

A data value of this data type is an attribute value whose syntax is `OM_S_INTEGER`.

OM_modification

The `OM_modification` data type is used for the *modification* parameter of **om_put( )**. Its C declaration is as follows:

```
typedef OM_uint OM_modification;
```

A data value of this data type indicates a kind of modification, as defined by **om_put( )**. It is chosen from the following set:

- `OM_INSERT_AT_BEGINNING`
- `OM_INSERT_AT_CERTAIN_POINT`
- `OM_INSERT_AT_END`
- `OM_REPLACE_ALL`
- `OM_REPLACE_CERTAIN_VALUES`

OM_object

The `OM_object` data type is used as a handle to either a private or a public object. Its C declaration is as follows:

```
typedef struct OM_descriptor_struct *OM_object;
```

A data value of this data type represents an object, which can be either public or private. It is an ordered sequence of one or more instances of the `OM_descriptor` data type. See the `OM_private_object` and `OM_public_object` data types for restrictions on that sequence ("OM_private_object" on page 245 and "OM_public_object" on page 246, respectively).

OM_object_identifier

The `OM_object_identifier` data type is used as an ASN.1 object identifier. Its C declaration is as follows:

```
typedef OM_string OM_object_identifier;
```

A data value of this data type contains an octet string that comprises the contents octets of the BER encoding of an ASN.1 object identifier.

C Declaration of Object Identifiers

Every application program that uses a class or another object identifier must explicitly import it into every compilation unit (C source module) that uses it. Each such class or object identifier name must be explicitly exported from just one compilation module. Most application programs find it convenient to export all the names they use from the same compilation unit. Exporting and importing is performed by using the following two macros:

- The importing macro makes the class or other object identifier constants available within a compilation unit.
  - `OM_IMPORT` *(class_name*)
  - `OM_IMPORT(` *OID_name*)
- The exporting macro allocates memory for the constants that represent the class or another object identifier.
  - `OM_EXPORT(` *class_name*)
  - `OM_EXPORT(` *OID_name*)

Object identifiers are defined in the appropriate header files, with the definition identifier having the prefix `OMP_O_` followed by the variable name for the object identifier. The constant itself provides the hexadecimal value of the object identifier string.

Use of Object Identifiers in C

The following macro initializes a descriptor:

```
OM_OID_DESC(
type, OID_name)
```

It sets the *type* component to that given, sets the *syntax* component to OM_S_OBJECT_IDENTIFIER_STRING, and sets the *value* component to the specified object identifier.

The following macro initializes a descriptor to mark the end of a client-allocated public object:

```
OM_NULL_DESCRIPTOR
```

For each class, there is a global variable of type OM_STRING with the same name; for example, the External class has a variable called OM_C_EXTERNAL. This is also the case for other object identifiers; for example, the object identifier for BER rules has a variable called OM_BER. This global variable can be supplied as a parameter to functions when required.

This variable is valid only when it is exported by an OM_EXPORT macro and imported by an OM_IMPORT macro in the compilation units that use it. This variable cannot form part of a descriptor, but the value of its length and elements components can be used. The following code fragment provides examples of the use of the macros and constants.

```
/* Examples of the use of the macros and constants */

#include <xom.h>

OM_IMPORT(OM_C_ENCODING)
OM_IMPORT(OM_CANONICAL_BER)

/*  The following sequence must appear in just one compilation
 *  unit in place of the above:
 *
 *  #include <xom.h>
 *
 *  OM_EXPORT(OM_C_ENCODING)
 *  OM_EXPORT(OM_CANONICAL_BER)
 */

main( )
{
/* Use #1 - Define a public object of class Encoding
 *         (Note: xxxx is a Message Handling class which
 *          can be encoded)
 */
OM_descriptor my_public_object[] = {
        OM_OID_DESC(OM_CLASS, OM_C_ENCODING),
        OM_OID_DESC(OM_OBJECT_CLASS, MA_C_xxxx),
        { OM_OBJECT_ENCODING, OM_S_ENCODING_STRING, \
          some_BER_value },
        OM_OID_DESC(OM_RULES, OM_CANONICAL_BER),
        OM_NULL_DESCRIPTOR
        };

/* Use #2 - Pass class Encoding as parameter to om_instance( )
 */
return_code = om_instance(my_object, OM_C_ENCODING,
&boolean_result);
}
```

OM_private_object

The OM_private_object data type is used as a handle to an object in an implementation-defined or private representation. Its C declaration is as follows:

```
typedef OM_object OM_private_object;
```

A data value of this data type is the designator or handle to a private object. It comprises a single descriptor whose *type* component is `OM_PRIVATE_OBJECT` and whose *syntax* and *value* components are unspecified.

Note: The descriptor's *syntax* and *value* components are essential to the service's proper operation with respect to the private object.

OM_public_object

The `OM_public_object` data type is used to define an object that can be directly accessed by a programmer. Its C declaration is as follows:

```
typedef OM_object OM_public_object;
```

A data value of this data type is a public object. It comprises one or more (usually more) descriptors, all but the last of which represent values of attributes of the object.

The descriptors for the values of a particular attribute with two or more values are adjacent to one another in the sequence. Their order is that of the values they represent. The order of the resulting groups of descriptors is unspecified.

Since the Class attribute specific to the Object class is represented among the descriptors, it must be represented before any other attributes. Regardless of whether or not the Class attribute is present, the syntax field of the first descriptor must have the `OM_S_SERVICE_GENERATED` bit set or cleared appropriately.

The last descriptor signals the end of the sequence of descriptors. The last descriptor's *type* component is `OM_NO_MORE_TYPES` and its *syntax* component is `OM_S_NO_MORE_SYNTAXES`. The last descriptor's *value* component is unspecified.

OM_return_code

The `OM_return_code` data type is used for a value that is returned from all OM functions, indicating either that the function succeeded or why it failed. Its C declaration is as follows:

```
typedef OM_uint OM_return_code;
```

A data value of this data type is the integer in the range 0 to 2^{16} that indicates an outcome of an interface function. It is chosen from the set specified in "XOM Return Codes" on page 252.

Integers in the narrower range 0 to 2^{15} are used to indicate the return codes they define.

OM_string

The `OM_string` data type is used for a data value of String syntax. Its C declaration is as follows:

```
typedef OM_uint32 OM_string_length;
typedef struct {
      OM_string_length length;
      void *elements;
} OM_string;

#define OM_STRING(string)\
      { (OM_string_length)(sizeof(string)-1), (string) }
```

A data value of this data type is a string; that is, an instance of a String syntax. A string is specified either in terms of its length or whether or not it terminates with NULL.

A string has the following components:

- *length* (`OM_string_length`)

  The number of octets by means of which the string is represented, or the `OM_LENGTH_UNSPECIFIED` value if the string terminates with NULL.
- *elements*

  The string's elements. The bits of a bit string are represented as a sequence of octets (see Figure 40). The first octet stores the number of unused bits in the last octet. The bits in the bit string, commencing with the first bit and proceeding to the trailing bit, are placed in bits 7 to 0 of the second octet. These are followed by bits 7 to 0 of the third octet, then by bits 7 to 0 of each octet in turn, followed by as many bits as are required of the final octet, commencing with bit 7.

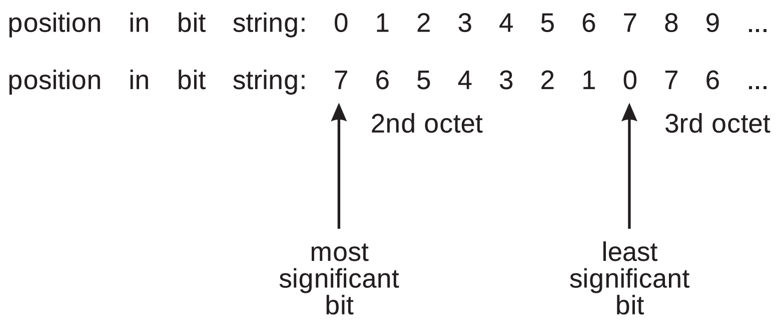

position in bit string: 0 1 2 3 4 5 6 7 8 9 ...
position in bit string: 7 6 5 4 3 2 1 0 7 6 ...
2nd octet
3rd octet
most significant bit
least significant bit

Figure 40. OM_String Elements

The service supplies a string value with a specified length. The client can supply a string value to the service in either form, either with a specified length or terminated with NULL.

The characters of a character string are represented as any sequence of octets permitted as the primitive contents octets of the BER encoding of an ASN.1 type value. The ASN.1 type defines the type of character string. A 0 (zero) value character follows the characters of the character string, but is not encompassed by the *length* component. Thus, depending on the type of character string, the 0 (zero) value character can delimit the characters of the character string.

The `OM_STRING` macro is provided for creating a data value of this data type, given only the value of its *elements* component. The macro, however, applies to octet strings and character strings, but not to bit strings.

OM_syntax

The `OM_syntax` data type is used to identify a syntax type. Its C declaration is as follows:

```
typedef OM_uint16 OM_syntax;
```

A data value of this data type is an integer in the range 0 to 2^9 that indicates an individual syntax or a set of syntaxes taken together.

The data value is chosen from among the following:
- `OM_S_BIT_STRING`
- `OM_S_BOOLEAN`
- `OM_S_ENCODING_STRING`
- `OM_S_ENUMERATION`
- `OM_S_GENERAL_STRING`
- `OM_S_GENERALIZED_TIME_STRING`
- `OM_S_GRAPHIC_STRING`
- `OM_S_IA5_STRING`
- `OM_S_INTEGER`
- `OM_S_NULL`
- `OM_S_NUMERIC_STRING`
- `OM_S_OBJECT`
- `OM_S_OBJECT_DESCRIPTOR_STRING`
- `OM_S_OBJECT_IDENTIFIER_STRING`
- `OM_S_OCTET_STRING`
- `OM_S_PRINTABLE_STRING`
- `OM_S_TELETEX_STRING`
- `OM_S_VIDEOTEX_STRING`
- `OM_S_VISIBLE_STRING`
- `OM_S_UTC_TIME_STRING`

Integers in the narrower range 0 to 2^9 are used to indicate the syntaxes they define. The integers in the range 2^9 to 2^{10} are reserved for vendor extensions. Wherever possible, the integers used are the same as the corresponding ASN.1 universal class number.

OM_type

The `OM_type` data type is used to identify an OM attribute type. Its C declaration is as follows:

```
typedef OM_uint16 OM_type;
```

A data value of this data type is an integer in the range 0 to 2^{16} that indicates a type in the context of a package. However, the following values in Table 62 on page 249 are assigned meanings by the respective data types.

Table 62. Assigning Meanings to Values

Value	Data Type
OM_NO_MORE_TYPES	OM_type_list
OM_PRIVATE_OBJECT	OM_private_object

Integers in the narrower range 0 to 2^{15} are used to indicate the types they define.

OM_type_list

The OM_type_list data type is used to enumerate a sequence of OM attribute types. Its C declaration is as follows:

```
typedef OM_type *OM_type_list;
```

A data value of this data type is an ordered sequence of zero or more type numbers, each of which is an instance of the OM_type data type.

An additional data value, OM_NO_MORE_TYPES, follows and thus delimits the sequence. The C representation of the sequence is an array.

OM_value

The OM_value data type is used to represent any data value. Its C declaration is as follows:

```
typedef struct {
      OM_uint32 padding;
      OM_object object;
} OM_padded_object;

typedef union OM_value_union {
      OM_string        string;
      OM_boolean       boolean;
      OM_enumeration   enumeration;
      OM_integer       integer;
      OM_padded_object object;
} OM_value;
```

Note: The first type definition (in particular, its padding component) aligns the object component with the *elements* component of the string component in the second type definition. This facilitates initialization in C.

The identifier OM_value_union is defined for reasons of compilation order. It is used in the definition of the OM_descriptor data type.

A data value of this data type is an attribute value. It has no components if the value's syntax is OM_S_NO_MORE_SYNTAXES or OM_S_NO_VALUE. Otherwise, it has one of the following components:

- string

  The value if its syntax is a string syntax
- boolean

  The value if its syntax is OM_S_BOOLEAN
- enumeration

  The value if its syntax is OM_S_ENUMERATION

- `integer`

  The value if its syntax is `OM_S_INTEGER`
- `object`

  The value if its syntax is `OM_S_OBJECT`

Note: A data value of this data type is only displayed as a component of a descriptor. Thus, it is always accompanied by indicators of the value's syntax. The latter indicator reveals which component is present.

OM_value_length

The `OM_value_length` data type is used to indicate the number of bits, octets, or characters in a string. Its C declaration is as follows:

```
typedef OM_uint32 OM_value_length;
```

A data value of this data type is an integer in the range 0 to 2^{32} that represents the number of bits in a bit string, octets in an octet string, or characters in a character string.

Note: This data type is not used in the definition of the interface. It is provided for use by client programmers for defining attribute constraints.

OM_value_position

The `OM_value_position` data type is used to indicate an attribute value's position within an attribute. Its C declaration is as follows:

```
typedef OM_uint32 OM_value_position;
```

A data value of this data type is an integer in the range 0 to 2^{32}-1 that indicates the position of a value within an attribute. However, the value `OM_ALL_VALUES` has the meaning assigned to it by **om_get( )**.

OM_workspace

The `OM_workspace` data type is used to identify an application-specific API that implements OM; for example, directory or message handling. Its C declaration is as follows:

```
typedef void *OM_workspace;
```

A data value of this data type is the designator or handle for a workspace.

XOM Functions

This section provides an overview of the XOM service interface functions as listed in Table 63. For a full description of these functions, see the `*(3xom)` reference pages.

Table 63. XOM Service Interface Functions

Function	Description
om_copy( )	Copies a private object.

Table 63. XOM Service Interface Functions (continued)

Function	Description
om_copy_value( )	Copies a string between private objects.
om_create( )	Creates a private object.
om_decode( )	Not supported by the DCE XOM interface; it returns an `OM_FUNCTION_DECLINED` error.
om_delete( )	Deletes a private or service-generated object.
om_encode( )	Not supported by the DCE XOM interface; it returns an `OM_FUNCTION_DECLINED` error.
om_get( )	Gets copies of attribute values from a private object.
om_instance( )	Tests an object's class.
om_put( )	Puts attribute values into a private object.
om_read( )	Reads a segment of a string in a private object.
om_remove( )	Removes attribute values from a private object.
om_write( )	Writes a segment of a string into a private object.

The purpose and range of capabilities of the service interface functions can be summarized as follows:

- **om_copy( )**

  This function creates an independent copy of an existing private object and all its subobjects. The copy is placed in the workspace of the original object, or in another workspace specified by the DCE client.
- **om_copy_value( )**

  This function replaces an existing attribute value or inserts a new value in one private object with a copy of an existing attribute value found in another. Both values must be strings.
- **om_create( )**

  This function creates a new private object that is an instance of a particular class. The object can be initialized with the attribute values specified as initial in the class definition. The service does not permit the client to explicitly create instances of all classes, but rather only those indicated by a package's definition as having this property.
- **om_decode( )**

  Not supported by the DCE XOM interface; it returns an `OM_FUNCTION_DECLINED` error.
- **om_delete( )**

  This function deletes a service-generated public object or makes a private object inaccessible.
- **om_get( )**

  This function creates a new public object that is an exact, but independent, copy of an existing private object. The client can request certain exclusions, each of which reduces the copy to a part of the original. The client can also request that values be converted from one syntax to another before they are returned.

The copy can exclude attributes of types other than those specified, values at positions other than those specified within an attribute, values of multivalued attributes, copies of (not handles for) subobjects, or all attribute values. Excluding all attribute values reveals only an attribute's presence.

- **om_instance( )**

  This function determines whether an object is an instance of a particular class. The client can determine an object's class simply by inspection. This function is useful since it reveals that an object is an instance of a particular class, even if the object is an instance of a subclass of that class.
- **om_put( )**

  This function places or replaces in one private object copies of the attribute values of another public or private object.

  The source values can be inserted before any existing destination values, before the value at a specified position in the destination attribute, or after any existing destination values. Alternatively, the source values can be substituted for any existing destination values or for the values at specified positions in the destination attribute.
- **om_read( )**

  This function reads a segment of a value of an attribute of a private object. The value must be a string. The value can first be converted from one syntax to another. This function enables the client to read an arbitrarily long value without requiring that the service place a copy of the entire value in memory.
- **om_remove( )**

  This function removes and discards particular values of an attribute of a private object. The attribute itself is removed if no values remain.
- **om_write( )**

  This function writes a segment of an attribute value to a private object. The value must be a string. The segment can first be converted from one syntax to another. The written segment becomes the value's last segment since any elements beyond it are discarded. The function enables the client to write an arbitrarily long value without having to place a copy of the entire value in memory.

XOM Return Codes

This section defines the return codes of the service interface, and thus the exceptions that can prevent the successful completion of an interface function.

Refer to the `ERRORS` section of the `*(3xom)` references pages for a list of the errors that each function can return. For an explanation of these error codes, refer to the *IBM DCE Version 3.1 for AIX and Solaris: Problem Determination Guide*.

The return code values are as follows:

0 `OM_SUCCESS`

Explanation: The function completed successfully.

1 `OM_ENCODING_INVALID`

Explanation: The octets that constitute the value of an encoding's Object Encoding attribute are invalid.

2 `OM_FUNCTION_DECLINED`

Explanation: The function does not apply to the object to which it is addressed.

3 `OM_FUNCTION_INTERRUPTED`

Explanation: The function is aborted by an external force. For example, a keystroke designated for this purpose at a user interface.

4 `OM_MEMORY_INSUFFICIENT`

Explanation: The service cannot allocate the main memory it needs to complete the function.

5 `OM_NETWORK_ERROR`

Explanation: The service could not successfully employ the network upon which its implementation depends.

6 `OM_NO_SUCH_CLASS`

Explanation: A purported class identifier is not defined.

7 `OM_NO_SUCH_EXCLUSION`

Explanation: A purported exclusion identifier is not defined.

8 `OM_NO_SUCH_MODIFICATION`

Explanation: A purported modification identifier is not defined.

9 `OM_NO_SUCH_OBJECT`

Explanation: A purported object is nonexistent, or the purported handle is invalid.

10 `OM_NO_SUCH_RULES`

Explanation: A purported rules identifier is not defined.

11 `OM_NO_SUCH_SYNTAX`

Explanation: A purported syntax identifier is not defined.

12 `OM_NO_SUCH_TYPE`

Explanation: A purported type identifier is not defined.

13 `OM_NO_SUCH_WORKSPACE`

Explanation: A purported workspace is nonexistent.

14 `OM_NOT_AN_ENCODING`

Explanation: An object is not an instance of the Encoding class.

15 `OM_NOT_CONCRETE`

Explanation: A class is abstract, not concrete.

16 `OM_NOT_PRESENT`

Explanation: An attribute value is absent, not present.

17 `OM_NOT_PRIVATE`

Explanation: An object is public, not private.

18 `OM_NOT_THE_SERVICES`

Explanation: An object is a client-generated object, rather than a service-generated or private object.

19 `OM_PERMANENT_ERROR`

Explanation: The service encountered a permanent difficulty other than those indicated by other return codes.

20 `OM_POINTER_INVALID`

Explanation: In the C interface, an invalid pointer is supplied as a function parameter, or as the receptacle for a function result.

21 `OM_SYSTEM_ERROR`

Explanation: The service could not successfully employ the operating system upon which its implementation depends.

22 `OM_TEMPORARY_ERROR`

Explanation: The service encountered a temporary difficulty other than those indicated by other return codes.

23 `OM_TOO_MANY_VALUES`

Explanation: An implementation limit prevents a further attribute value from being added to an object. This limit is undefined.

24 `OM_VALUES_NOT_ADJACENT`

Explanation: The descriptors for the values of a particular attribute are not adjacent.

25 `OM_WRONG_VALUE_LENGTH`

Explanation: An attribute has, or would have, a value that violates the value length constraints in force.

26 `OM_WRONG_VALUE_MAKEUP`

Explanation: An attribute has, or would have, a value that violates a constraint on the value's syntax.

27 `OM_WRONG_VALUE_NUMBER`

Explanation: An attribute has, or would have, a value that violates the value number constraints in force.

28 `OM_WRONG_VALUE_POSITION`

Explanation: The use defined for value position in the parameter or parameters of a function is invalid.

29 `OM_WRONG_VALUE_SYNTAX`

Explanation: An attribute has, or would have, a value whose syntax is not permitted.

30 `OM_WRONG_VALUE_TYPE`

Explanation: An object has, or would have, an attribute whose type is not permitted.

Chapter 14. Object Management Package

This chapter defines the object management package (OMP). The object identifier (referred to as om) assigned to the package, as defined by this guide, is the object identifier specified in ASN.1 as

```
{joint-iso-ccitt(2) mhs-motis(6) group(6) white(1) api(2) om(4)}
```

Class Hierarchy

This section shows the hierarchical organization of the OM classes. Subclassification is indicated by indentation, and the names of abstract classes are in italics. Thus, for example, OM_C_ENCODING is an immediate subclass of *OM_C_OBJECT*, an abstract class. The names of classes to which **om_encode( )** applies are in boldface. (DCE XOM does not support the encoding of any OM classes.) The **om_create( )** function applies to all concrete classes.

- *OM_C_OBJECT*
  - OM_C_ENCODING
  - OM_C_EXTERNAL

Class Definitions

The following subsections define the OM classes.

OM_C_ENCODING

An instance of class OM_C_ENCODING is an object represented in a form suitable for transmission between workspaces, for transport via a network, or for storage in a file. Encoding can also be a suitable way of indicating to an intermediate service provider (for example, a directory, or message transfer system) an object that it does not recognize.

This class has the attributes of its superclass, *OM_C_OBJECT*, in addition to the specific attributes listed in Table 64.

Table 64. Attributes Specific to OM_C_ENCODING

Attribute	**Value Syntax**	**Value Length**	**Value Number**	**Value Initially**
OM_OBJECT_ CLASS	String(OM_S_ OBJECT_ IDENTIFIER_ STRING)	—	1	—
OM_OBJECT_ ENCODING	String[1]	—	1	—
OM_RULES	String(OM_S_ OBJECT_ IDENTIFIER_ STRING)	—	1	ber

1 If the Rules attribute is ber or canonical-ber, the syntax of the present attribute must be String(OM_S_ENCODING_STRING).

- OM_OBJECT_CLASS

  This attribute identifies the class of the object that the Object Encoding attribute encodes. The class must be concrete.
- OM_OBJECT_ENCODING

  This attribute is the encoding itself.

- OM_RULES

  This attribute identifies the set of rules that are followed to produce the Object Encoding attribute. Among the defined values of this attribute are those represented as follows:

  – OM_BER

    This value is specified in ASN.1 as

    ```
    {joint-iso-ccitt(2) asn1(1) basic-encoding(1)}
    ```

    This value indicates the BER. (See Clause 25.2 of Recommendation X.209, ″Specification of Basic Encoding Rules for Abstract Syntax Notation 1 (ASN.1),'' *CCITT Blue Book*, Fascicle VIII.4, International Telecommunications Union, 1988. Also published by ISO as *ISO 8825*.)

  – OM_CANONICAL_BER

    This value is specified in ASN.1 as

    ```
    {joint-iso-ccitt(2) mhs-motis(6) group(6) white(1) api(2) om(4)
    canonical-ber(4)}
    ```

    This value indicates the canonical BER. (See Clause 8.7 of Recommendation X.509, ″The Directory: Authentication Framework,'' *CCITT Blue Book,* International Telecommunications Union, 1988. Also published by ISO as *ISO 9594-8*.)

Note: In general, an instance of this class cannot appear as a value whose syntax is Object (*C*) if *C* is not OM_C_ENCODING, even if the class of the object encoded is *C*.

OM_C_EXTERNAL

An instance of class OM_C_EXTERNAL is a data value and one or more information items that describe the data value and identify its data type. This class corresponds to ASN.1's External type, and thus the class and the attributes specific to it are described indirectly in the specification of ASN.1. (See Clause 34 of Recommendation X.208, ″Specification of Abstract Syntax Notation 1 (ASN.1),'' *CCITT Blue Book*, Fascicle VIII.4, International Telecommunications Union, 1988. Also published by ISO as *ISO 8824*.)

This class has the attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attributes specific to this class that are listed in Table 65.

Table 65. Attributes Specific to OM_C_EXTERNAL

Attribute	**Value Syntax**	**Value Length**	**Value Number**	**Value Initially**
OM_ ARBITRARY_ ENCODING	String(OM_S_ BIT_STRING)	—	0 or 1[1]	—
OM_ASN1_ ENCODING	String(OM_S_ ENCODING_ STRING)	—	0 or 1[1]	—
OM_DATA_ VALUE_ DESCRIPTOR	String(OM_S_ OBJECT_ DESCRIPTOR_ STRING)	—	0 or 1	—
OM_DIRECT_ REFERENCE	String(OM_S_ OBJECT_ IDENTIFIER_ STRING)	—	0 or 1	—
OM_INDIRECT_ REFERENCE	OM_S_ INTEGER	—	0 or 1	—

Table 65. Attributes Specific to OM_C_EXTERNAL (continued)

Attribute	Value Syntax	Value Length	Value Number	Value Initially
OM_OCTET_ ALIGNED_ ENCODING	String(OM_S_ OCTET_ STRING)	—	0 or 1[1]	—

[1] Only one of these three attributes is present.

- OM_ARBITRARY_ENCODING

  This attribute is a representation of the data value as a bit string.
- OM_ASN1_ENCODING

  The data value. This attribute can be present only if the data type is an ASN.1 type.

  If this attribute value's syntax is an Object syntax, the data value's representation is that produced by om_encode( ) when its *Object* parameter is the attribute value and its *Rules* parameter is ber. Thus, the object's class must be one to which om_encode( ) applies.
- OM_DATA_VALUE_DESCRIPTOR

  This attribute contains a description of the data value.
- OM_DIRECT_REFERENCE

  This attribute contains a direct reference to the data type.
- OM_INDIRECT_REFERENCE

  This attribute contains an indirect reference to the data type.
- OM_OCTET_ALIGNED_ENCODING

  This attribute contains a representation of the data value as an octet string.

OM_C_OBJECT

The class *OM_C_OBJECT* represents information objects of any variety. This abstract class is distinguished by the fact that it has no superclass and that all other classes are its subclasses.

The attribute specific to this class is listed in Table 66.

Table 66. Attribute Specific to OM_C_OBJECT

Attribute	Value Syntax	Value Length	Value Number	Value Initially
OM_CLASS	String(OM_S_ OBJECT_ IDENTIFIER_ STRING)	—	1	—

- OM_CLASS

  This attribute identifies the object's class.

Appendix. Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the information. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this information at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs

Trademarks

Index

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.