VMS Software

# VSI X.500 Directory Service Programming

**Operating System and Version:** VSI OpenVMS Alpha Version 8.4-2L1 or higher
VSI OpenVMS IA-64 Version 8.4-1H1 or higher

**VMS Software, Inc. (VSI)**
**Boston, Massachusetts, USA**

# VSI X.500 Directory Service Programming

VMS Software



## Legal Notice



Confidential computer software. Valid license from VSI required for possession, use or copying. Consistent with FAR 12.211 and 12.212, Commercial Computer Software, Computer Software Documentation, and Technical Data for Commercial Items are licensed to the U.S. Government under vendor's standard commercial license.

The information contained herein is subject to change without notice. The only warranties for VSI products and services are set forth in the express warranty statements accompanying such products and services. Nothing herein should be construed as constituting an additional warranty. VSI shall not be liable for technical or editorial errors or omissions contained herein.

# Preface

## 1. About VSI

VMS Software, Inc. (VSI) is an independent software company licensed by Hewlett Packard Enterprise to develop and support the OpenVMS operating system.

## 2. Purpose of This Guide

This guide helps you build an application that uses VSI X.500 Directory Service Application Programming Interface (API), XDS. You can use XDS in X.500 environments and DCE environments. For information about using XDS and the DCE Cell Directory Service (CDS), see the *DCE for DEC™ OSF/1 AXP™ Product Guide*.

## 3. Document Structure

This document contains the following chapters:

- Chapter 1
- Chapter 2
- Chapter 3
- Appendix A

## 4. VSI Encourages Your Comments

You may send comments or suggestions regarding this manual or any VSI document by sending electronic mail to the following Internet address: <docinfo@vmssoftware.com>. Users who have VSI OpenVMS support contracts through VSI can contact <support@vmssoftware.com> for help with this product.

## 5. OpenVMS Documentation

The full VSI OpenVMS documentation set can be found on the VMS Software Documentation webpage at https://docs.vmssoftware.com.

## 6. Prerequisites

Before you can develop an application that uses the X.500 interface, you need to be familiar with X.500 terms and concepts, as explained in *DEC X.500 Directory Service Management*. The XDS component of the VSI X.500 Directory Service must be installed.

You must be familiar with the C programming language. This is the programming language that the X.500 API supports.

## 7. Related Documents

Before you write an X.500 application, you should have available the following books:

- *VSI X.500 Directory Service Programming Reference*
- *DEC X.500 Directory Service Management*
- *DEC X.500 Directory Service Problem Solving*
- *OSI-Abstract-Data Manipulation*

If you are writing a CDS application, refer to *DCE for DEC OSF/1 AXP Product Guide*.

# 8. Conventions

The term X.500 API is used to describe XDS. For details of the XDS interface and DCE CDS, refer to the DCE documentation.

The following conventions are used in this book:

| Convention | Meaning |
|---|---|
| `this typeface` | Indicates an example of code |
| ***bold italics*** | Indicates new terminology |
| *italic text* | Indicates attribute values |
| OpenVMS™ | Means both OpenVMS AXP and OpenVMS VAX™ systems, unless otherwise stated |

# Chapter 1. Planning Your Application

VSI X.500 API is an implementation of Version 1 of the X/Open™CAE Specification *API to Directory Services (XDS)* . This specification defines an application programming interface to an Open Systems Interconnection (OSI) Directory Service, X.500. XDS is also the interface to the Cell Directory

Service (CDS) component of the Distributed Computing Environment (DCE).

This book describes how to use XDS to write an X.500 application, and uses the term ***X.500 API*** to refer to XDS. For information about X.500 and VSI X.500 Directory Service, refer to *DEC X.500 Directory Service Management* . For information about the DCE products and how to use XDS to write a CDS application, see *DCE for DEC OSF/1 AXP Product Guide*.

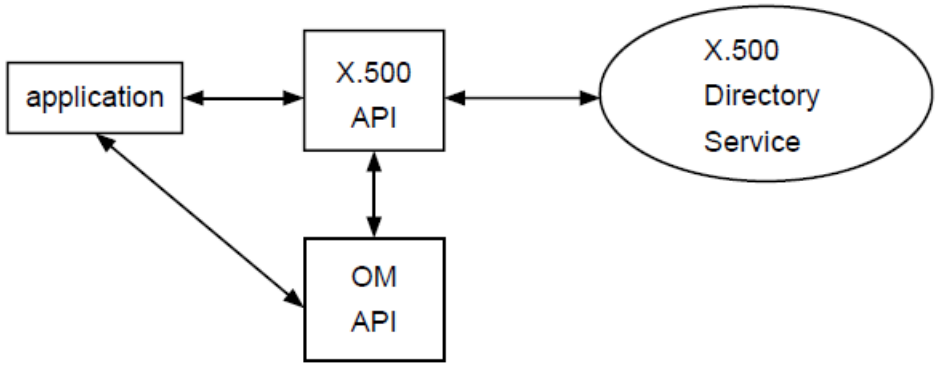
The X.500 API uses the OSI-Abstract-Data Manipulation (OM) API to store and manipulate objects. See *OSI-Abstract-Data Manipulation* for details of the OM API.

This chapter explains how to plan an application. To do this, you need to understand:

- How the X.500 API uses the X.500 Directory Service
- How your application uses the X.500 API
- How your application and the X.500 API use the OM API

Figure 1.1 shows the conceptual model of the interaction between an application, the X.500 API, the OM API, and the X.500 Directory Service.

**Figure 1.1. Interfaces to the X.500 API**




Section 1.1explains how the X.500 API uses the services provided by the X.500 Directory Service and how you can use them.

Section 1.2 explains how to plan what information needs to be passed between the X.500 API and the X.500 Directory Service, and the format of that information.

Section 1.3explains how to plan the data structures used to pass the information between your application and the X.500 API. This section also includes information on how to use the OM API.

# 1.1. Assessing Your Requirements

The X.500 Directory Service provides services to allow interrogation and modification of information stored in the DIB. The X.500 API provides functions that allow applications to use these services. Each function maps to one service, as described in *VSI X.500 Directory Service Programming Reference*.

You need to understand what each of the X.500 API functions do. You must also understand how information is stored in the X.500 directory and the effect of different parameters and parameter values on the operations that you invoke.

## 1.1.1. Assessing the Services

When you are preparing to write an application, you need to consider which of the functions must be used by all applications, and which suit the needs of your application. The functions are divided into three categories:

- Administration
- Interrogation
- Modification

The administration functions are concerned with the opening and closing of communications between the application, the X.500 API, and the X.500 Directory Service. The administration functions are:

- Initialize, which initializes the interface and establishes the connection between the application and API by setting up a workspace.
- Bind, which opens a session with the X.500 Directory Service and specifies which DSA the application wants, by default, to communicate with.
- Version, which is an optional function that negotiates the use of standard or vendor extensions to the interface.
- Unbind, which closes a session with the X.500 Directory Service.
- Shutdown, which shuts down the interface and releases the workspace.

All administration functions, except for the Version function, are needed to open and close a session between the application and the X.500 API.

The interrogation functions are:

- Read, which reads entry information.
- List, which lists immediate subordinate entries of a specified entry.
- Compare, which compares entry attributes with specified values.
- Search, which searches for entries.

The modification functions are:

- Add-Entry, which adds a new entry to the Directory.
- Modify-Entry, which modifies an existing entry.

- Remove-Entry, which deletes an existing entry.
- Modify-RDN, which modifies the Relative Distinguished Name (RDN) of an existing entry.

Two further functions are provided for asynchronous operations; these are:

- Receive-Result, which is used to return the results of an outstanding asynchronous operation.
- Abandon, which is used to abandon an outstanding asynchronous operation. This causes any results to be ignored or discarded.

See Chapter 2 for details of these functions and how to use them.

You need a complete understanding of the purpose of your application in order to understand whether, for example, it needs to use the Compare function. Some applications use the interrogation functions, others use the modification functions, and many use a combination of both.

### 1.1.2. Using the Services

You need to consider how to use the functions provided by the X.500 API to satisfy the requirements of your application.

When you are planning to use interrogation functions, you need to consider the following points:

- Which entries you need to interrogate.
- Which entry attribute types and attribute values you want the operation to return.
- Which method of interrogation you must use to obtain the required information. For example, can you just Read a specified entry or do you need to perform a Search to find the Distinguished Name of the entry? For a full description of how you can interrogate entries, see Section 2.5.
- Why you need the operation. This is essential for determining how accurate and complete the results from the operation must be. This requirement will decide which Context parameters you pass to the function. For a full explanation of this, see Section 1.4.

When you are planning to use modification functions, you need to consider the following:

- Which entry or entries need to be modified.
- What modifications are to be made on each entry. For details of how you can modify entries, see Section 2.6.

Each call to an X.500 API function invokes a corresponding directory operation. The parameters and parameter values that you pass to the function determine how the function processes that operation. Consider how to use these parameters to make these operations as effective and efficient as possible, obtaining the required results with the minimum use of resources. Section 1.4 explains how to achieve this using the Context object.

## 1.2. Researching the Information Requirements of Your Application

Before developing an application, you must consider what information the application needs to store in, and retrieve from, the DIB. The information requirements of an application might be unique to that application.

You also need to understand how the application presents information to the X.500 Directory Service, and how it receives information. The X.500 Directory Service passes data across the X.500 API using a standard syntax, which is described in *DEC X.500 Directory Service Programming Reference* . An application must be able to make use of the data in that syntax or else be able to convert it into a usable form. Similarly, all data passed to the X.500

Directory Service must be in a defined syntax, and an application must be able to present the data in that syntax.

Information is passed between the X.500 API and the X.500 Directory Service using directory objects. The CCITT X.500 Series of Recommendations defines a set of object classes, attribute types, and attribute syntaxes for directory objects.

An application can use the X.500 API to gain access to any data held anywhere in the Directory, not just that held at the DSA you bind to. You therefore need to understand the characteristics and limitations of the Directory Service as a whole.

You must plan an application's storage of information in the DIB carefully, considering the needs of other applications and the object and attribute definitions. It is possible that the object classes and attribute types provided by the X.500 API are sufficient for the needs of your application. However, it is also possible that you will need to define new object classes and attribute types. This planning task is explained more fully in the *X.500 Directory Service Management Guide*.

# 1.3. Data Structures

You need to plan how your application is going to store data that either it creates or has received from the X.500 API. You need to consider the data structures required to store the data, and the amount of memory, or disk space, that it occupies. The guide *OSI-Abstract-Data Manipulation* contains more information about this.

## 1.3.1. Categories of Data

An application may need one or both of the following categories of data:

- **Static data**; that is, data whose structure is fixed by the application when you compile it, for example, an object that is to contain a Distinguished Name.

- **Dynamic data** ; that is, data whose structure is not fixed by the application but must be defined when you run the application. For example, you define a search filter but the application user may specify any number of search criteria to be included in the filter.

You need to consider the different storage requirements for both categories of data. Either category can be stored by the application to be passed into the X.500 API.

The structure of static data is determined when you compile the application. This structure cannot be altered when you run the application. The way the data is to be stored, and the amount of storage it requires, is determined when you compile the application. The only changes that can be made are to the data values.

The structure of dynamic data must be defined when you compile the application but is not fixed. The way the data is stored, and the amount of storage it requires can alter while the application is running. Therefore, the dynamic nature of the data structure must be planned for. Typically, this type of data is

used most often in an interactive application, hence these considerations would be involved in the design of a user interface.

Data is exchanged between the application and the X.500 API using objects defined by Object Management (OM). The OM objects and attributes represent the service elements and parameters used in the DAP protocol.

The information that an OM object contains is defined by the value of the Object Class attribute. The type of an OM object defines how it can be manipulated.

The following types of object are defined by OM:

- Private objects
- Client-generated public objects
- Service-generated public objects

These objects are described in Section 1.3.2.

## 1.3.2. Private and Public Objects

A ***private object*** has a format that is known only to the X.500 API and not to an application. To create a private object, you use the OM functions. You therefore build a private object dynamically, when the application is run; you cannot declare it statically.

Private objects are created and maintained by the X.500 API in a workspace. A workspace is an area of memory that is a repository for private objects. The workspace has object class ***packages*** associated with it to enable classes within the packages to be created. A package is a group of OM classes that are functionally related.

You manipulate private objects using OM functions, which are described briefly in Section 2.2. See *OSI-Abstract-Data Manipulation* for further information on private objects, public objects, and the OM functions.

A ***public object*** consists of a data structure called a descriptor list, as described in Section 1.3.3. The structure of a public object is known to an application. It contains a collection of attributes determined by the OM object class. Public objects are data structures that can be declared statically or built dynamically. See Section 2.2 for details of how to do this.

There are two types of public object:

- Client-generated

  These are public objects created by your application in storage that it allocates. You use client-generated public objects to represent information that is passed to the X.500 API.

- Service-generated

  These are public objects created by the X.500 API in storage that it allocates. Service-generated public objects represent information that is returned by the X.500 API to the application. They cannot be modified by the application.

When you call an X.500 API function, you can specify private or public objects for many of the parameters. Passing a private object to a function may be more efficient because if you pass a public

object, the X.500 API must convert it to a private object before it can access the Directory. If you are passing the information to the X.500 API once and do not need to manipulate the object, then use a public object. If you need to manipulate the object, using the OM functions, then use a private object.

If the data must be built dynamically, then do either of the following:

- Build a public object.
- Use the OM-Create and then the OM-Put functions to build a private object.

## 1.3.3. Descriptor Structures

The X.500 API defines data structures that are used by applications to store all the required data. You need to understand these data structures to plan data storage. These data structures are descriptors and descriptor lists.

A ***descriptor*** is a defined data structure that represents a single OM attribute. The structure has three components:

- A type that identifies the type of the attribute that the descriptor represents.
- A syntax that identifies the syntax associated with the attribute type that the descriptor represents. This tells you the format of the information to enable you to access it.
- A value that identifies the value of the attribute that the descriptor represents. This can be one of the following:
    - The information stored in the attribute, if the type is an integer, enumeration or Boolean.
    - A pointer to the information, if the type is an object or a string.

Section 1.3.3 shows how a descriptor is constructed. The example programs described in Section 1.9 illustrate how such structures can be used.

A ***descriptor list*** is an ordered sequence of descriptors that can represent several OM attribute types and values in a single OM object. Descriptor lists can store single-valued and multi-valued OM attributes.

A single-valued OM attribute is represented by a single descriptor in a descriptor list. A multi-valued OM attribute is represented by several descriptors with the same attribute type and syntax in a descriptor list. The descriptors representing the values of a multi-valued OM attribute are always adjacent in the descriptor list. The order of the values in the OM attribute is the same as the order in the descriptor list.

A single descriptor list can represent several single-valued and multi-valued OM attributes.

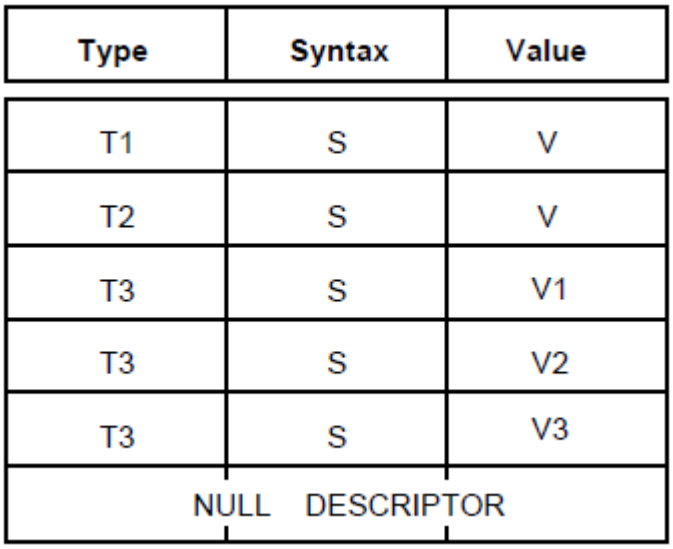
If a descriptor list contains a descriptor representing an OM class, then that must be the first descriptor in the descriptor list. A descriptor list is terminated with a null descriptor.

Figure 1.2 represents a descriptor list containing the following descriptors:

- A class descriptor, T1, representing the class of the public object.
- A single descriptor, T2, representing a single-valued attribute in the public object.
- Three descriptors, T3, representing an attribute with three values in the public object.

- The null descriptor indicating the end of the descriptor list.

**Figure 1.2. Components of a Descriptor List**



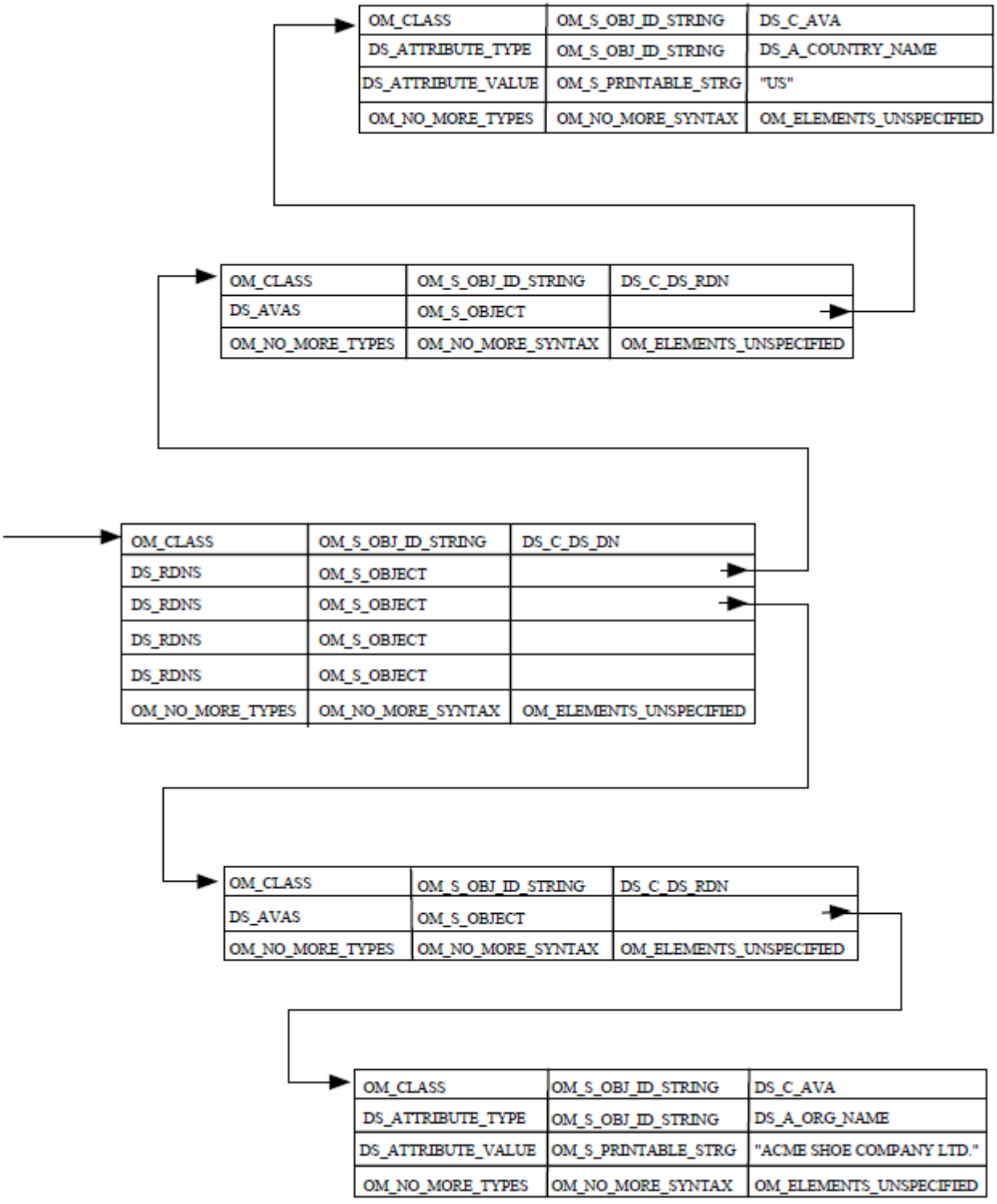

The following code is an example of coding a descriptor list to represent a distinguished name. The structure macros are defined in the file `xom.h`.

```
/**** Public Object ("Descriptor List") for NAME argument to READ*****/
 OM_descriptor name[] =
 {
 OM_OID_DESC(OM_CLASS, DS_C_DS_DN),
 { DS_RDNS, OM_S_OBJECT, { 0, DIB_RDN_country_US } },
 { DS_RDNS, OM_S_OBJECT, { 0, DIB_RDN_org_name_ABACUS_CO } },
 { DS_RDNS, OM_S_OBJECT, { 0, DIB_RDN_locality_USA } },
 { DS_RDNS, OM_S_OBJECT, { 0, DIB_RDN_org_unit_SALES } },
 OM_NULL_DESCRIPTOR
 };
 /**** Country ********************************************************/
 OM_descriptor DIB_country_US[] =
 {
 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
 };
 OM_descriptor DIB_RDN_country_US[] =
 {
 OM_OID_DESC(OM_CLASS, DS_C_DS_RDN),
 { DS_AVAS, OM_S_OBJECT, { 0, DIB_country_US } },
 OM_NULL_DESCRIPTOR
 };
 /**** Organizations **************************************************/
 OM_descriptor DIB_org_name_ABACUS_CO[] =
 {
 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("ABACUS LTD.") },
 OM_NULL_DESCRIPTOR
```

```
};
OM_descriptor DIB_RDN_org_name_ABACUS_CO[] =
{
OM_OID_DESC(OM_CLASS, DS_C_DS_RDN),
{ DS_AVAS, OM_S_OBJECT, { 0, DIB_org_name_ABACUS_CO } },
OM_NULL_DESCRIPTOR
};
/**** Locality Names ************************************************/
OM_descriptor DIB_locality_USA[] =
{
OM_OID_DESC(OM_CLASS, DS_C_AVA),
OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_LOCALITY_NAME),
{ DS_ATTRIBUTE_VALUES, OM_S_PRINTABLE_STRING, OM_STRING("USA") },
OM_NULL_DESCRIPTOR
};
OM_descriptor DIB_RDN_locality_USA[] =
{
OM_OID_DESC(OM_CLASS, DS_C_DS_RDN),
{ DS_AVAS, OM_S_OBJECT, { 0, DIB_locality_USA } },
OM_NULL_DESCRIPTOR
};
/**** Organizational Units ******************************************/
OM_descriptor DIB_org_unit_SALES[] =
{
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("Sales") },
OM_NULL_DESCRIPTOR
};
OM_descriptor DIB_RDN_org_unit_SALES[] =
{
OM_OID_DESC(OM_CLASS, DS_C_DS_RDN),
{ DS_AVAS, OM_S_OBJECT, { 0, DIB_org_unit_SALES } },
OM_NULL_DESCRIPTOR
};
```

Figure 1.3 shows the relationship between the descriptor list and its descriptors.

**Figure 1.3. Relationship Between Descriptors and Descriptor Lists**




## 1.3.4. Using Object Management Objects

When an object of a particular OM class is required, an instance of any subclass of that class can be supplied instead. This means that the application can supply a subclass to the X.500 API, and the X.500 API can return a subclass to the application. For example, the application can supply entry names in any format that is defined as a subclass of the OM class Name, and the X.500 API returns all errors as a

subclass of the OM class Error. See *DEC X.500 Directory Service Programming Reference* for details of the classes and subclasses defined.

Applications should not check the OM class of an object by requesting the value of its OM attribute Class. Applications must always use the OM-Instance function, which inform s you whether an object is an instance of a particular OM class or any of its subclasses.

# 1.4. Controlling Directory Operations

The X.500 Directory Service allows you to place certain controls and constraints on operations invoked by the X.500 API functions. This enables you to control aspects of the X.500 Directory Service operation to suit your requirements.

You can specify the controls you require as an object of the OM class **Context** , which is passed as the second argument of every X.500 API interrogation or modification function. The attributes of this object are like extra parameters to the function and determine the following:

- Whether an operation is processed synchronously or asynchronously.
- Whether an operation is distributed around the Directory and how far it can be distributed.
- What limits are to be placed on the operation. You can place limits on the time the operation takes and on the amount of information it returns.
- The level of service required by the operation. This includes the setting of priority for operations, and whether or not alias and copy entries are used.

You can change the context from operation to operation, but normally the same context suits several consecutive operations or even an entire session.

You must consider how the use of different context parameters affect s the results returned by the X.500 API functions. You also need to consider how it affect s the user's view of your application. For example, placing a time limit on operations could mean that a function returns only partial results but can also save the user unnecessary waiting.

The following sections explain the Context OM object parameters and their defaults. Section 1.4.9 explains how use the system default settings for the context parameters, and how to use the DUA defaults file to override the system defaults. *VSI X.500 Directory Service Programming Reference* contains a full description of the Context OM object and the default values of its attributes.

## 1.4.1. Choosing Synchronous and Asynchronous Operations

This section describes synchronous and asynchronous modes and explains what you need to consider when choosing between them. It also describes what extra planning is required for asynchronous operations.

### 1.4.1.1. Synchronous Operations

***Synchronous mode*** means that all the functions send a request to the Directory and wait until the operation is complete before returning control to the caller, with the result or an error. This allows the application to use one call at a time and to receive the result as soon as the directory has processed the operation. It also means that the thread of control of the application is blocked from the time a function is called until the result is returned.

All errors occurring during a synchronous operation are reported when the function returns.

Use synchronous mode when:

- The sequence in which the results are returned is important. Asynchronous mode does not guarantee to return the results in any particular sequence.
- An application uses the directory infrequently or can do no useful work until it receives the result of a given operation.

To have an operation executed synchronously, use the default value of *false* for the asynchronous parameter.

### 1.4.1.2. Asynchronous Operations

***Asynchronous mode*** means that the execution of operations does not block the thread of control. Functions send a request to the Directory and return before the operation completes. The application receives the DSA's response to the request at some later time. Therefore, the application can make many concurrent asynchronous calls in the same session. Responses are not returned in any particular order.

Version 1 of the X/Open CAE Specification *API to Directory Services (XDS)* states that the Library-Error *mixed-synchronous* , may be returned if there are any outstanding asynchronous operations when a synchronous call is made. VSI X.500 API allows synchronous calls to be made even when there are outstanding asynchronous operations. However, if there are outstanding asynchronous operations, you must call the Receive-Result function after each synchronous call. You do not need to do this if your application polls for results of asynchronous operations by calling the Receive-Result function regularly. Refer to Appendix A for more information.

There is a limit to the the number of outstanding asynchronous operations allowed at any one time during a session. This is determined by the constant Max-Outstanding-Operations. This constant, DS_MAX_OUTSTANDING_ OPERATIONS, is defined in the header file `xds.h`.

An asynchronous operation is outstanding from the time the function is called until the result is returned by Receive-Result or abandoned by the Abandon function. If the number of outstanding operations equals this limit, any attempt to invoke further asynchronous operations will return the Library- Error, *too-many-operations* .

If an error is detected before an asynchronous request is submitted to the directory, the function returns immediately and no outstanding operation is generated. Other errors are reported when the results of the operation are returned by the Receive-Result function.

Applications must ensure that there are no outstanding operations in a session when a call to Unbind is made on that session. After Unbind is called, you cannot determine whether any outstanding operations succeeded or whether they were sent to the directory. No results or errors of any kind are returned in this situation.

The reasons for using asynchronous operations are:

- The application can potentially be doing other useful work while it waits for results from function calls.
- The application can initiate several operations in quick succession, then collect the results when required.
- If an error is detected before an operation is submitted to the directory, the function will return immediately. No outstanding operation will be generated, thus saving time.

To have an operation executed asynchronously, set the asynchronous parameter to *true*.

## 1.4.2. Limiting Operations

The cost of some directory operations in both time and resources could become high if they are not limited in some way. Therefore, you can limit the amount of information returned by an operation or the amount of processing time devoted to it. For example, you may need to limit a search operation that could take a long time, return large amounts of data, or both.

### 1.4.2.1. Limiting Information Returned

If you want to set a maximum number of objects on which List or Search operations return information, use the Size-Limit parameter. This enables you to limit the amount of information returned to suit your resources of time and available storage space.

Choose this option if you know that your application can only store information about a limited number of objects or if you require information from a small sample of objects.

The default value is *zero*, indicating no limit.

### 1.4.2.2. Limiting Processing Time

If you want to set a maximum amount of time that the X.500 Directory Service spends processing an operation, use the Time-Limit parameter. You specify the limit in seconds, indicating the maximum elapsed time between submitting the request and the operation completing. If this limit is reached then the X.500 Directory Service terminates the operation and returns the results accumulated so far to the X.500 API.

Choose this option if you know that the operation you have invoked could take a long time to return and you are not prepared to wait for more than a certain length of time. For example, for an interactive application, a small sample of results from a search operation could contain the required information; the user does not need to wait for the whole search operation to complete. In this case, the user could choose to repeat the search for a longer time if the operation did not return the required information. Limiting processing time can prevent users submitting expensive searches.

The default value is *zero*, indicating no limit.

## 1.4.3. Distributing Requests

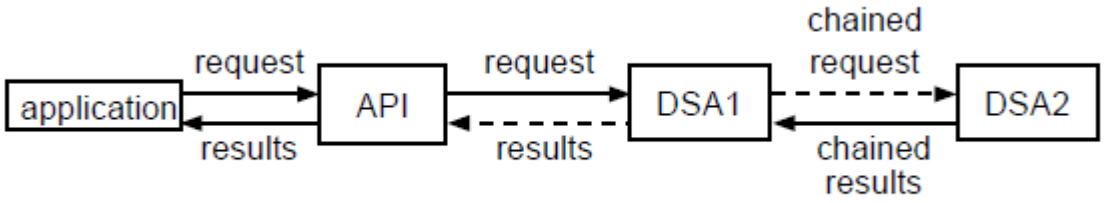
When a DSA is sent a request by the X.500 API but does not hold the necessary information, it can do one of the following:

- Return a referral to the X.500 API indicating a DSA or DSAs that might satisfy the request. See Section 1.4.4 for details of referrals.
- Distribute, by chaining, the request to one or more other DSAs.

***Chaining*** is the sending of a request by a DSA to another DSA. It is possible for a request to be chained to any DSA that is known to the DSA that chains the request. A request can be chained to one DSA or to several DSAs simultaneously. Also, a DSA that receives a chained request can chain it to other DSAs.

Figure 1.4 shows a request chained to a single DSA. Figure 1.5 shows a request chained to several DSAs simultaneously.

**Figure 1.4. Chaining to a Single DSA**

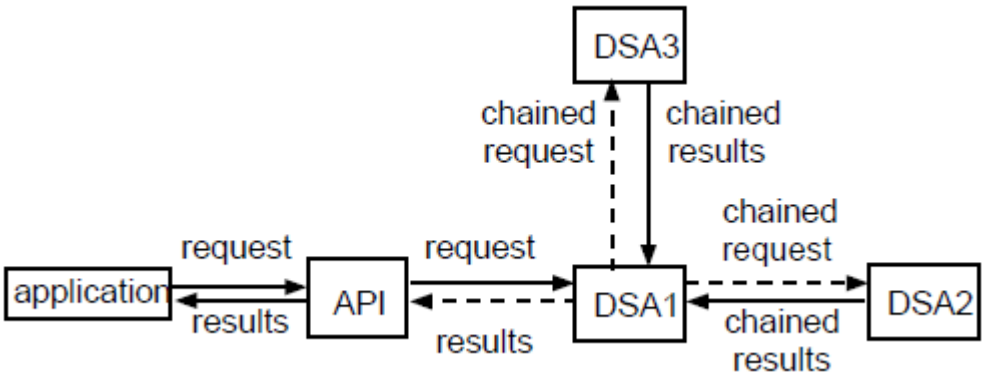



Use the Chaining-Prohibited parameter to control whether requests are permitted to be chained to other DSAs, or the operation can only return information from the portion of the DIB held by the DSA to which the request was originally sent. Set the Chaining-Prohibited parameter as follows:

- To prevent chaining, set Chaining-Prohibited to *true*.

  Use this value if you need to minimize the use of resources. For example, for an interactive application, you could stop operations being chained by default but allow the user to invoke operations with the Chaining- Prohibited parameter set to *false* .

  **Figure 1.5. Simultaneous Chaining to Multiple DSAs**




- To permit chaining, use the default value of *false* for Chaining-Prohibited.

  Use this value when it is important that the operation returns the required information. For example, the application might require certain information to enable it to produce an accurate management report.

Use the Prefer-Chaining parameter to specify whether or not you prefer a request to be chained, instead of a referral returned, when the DSA cannot satisfy it. Select your preference as follows:

- If you prefer chaining, then use the default value of *true* for Prefer-Chaining. Note that this control does not force the request to be chained; the Directory can still return a referral if for some reason the request cannot be chained.

- If you have no preference, then set Prefer-Chaining to *false* .

You should not set both Prefer-Chaining and Chaining-Prohibited to *true*, but you can set both to *false* .

Use the Local-Scope parameter to specify whether or not requests can be chained outside of an implementation-defined local scope. The local scope will typically be the ***Directory Management***

***Domain*** (DMD) or DSA in which the request originates. If possible, you should check the definition of the local scope parameter for the DSA that you are using. Set this parameter as follows:

- If you do not want requests to be chained outside the local scope, then use the default value of *true* for Local-Scope.
- If you want requests chained outside of the local scope, then set Local- Scope to *false* . This means that a request could be chained to any DSA, anywhere in any directory management domain.

A DSA defines local scope to be the DSA in which the request originates, effectively preventing chaining when the value of Local-Scope is *true*.

## 1.4.4. Returning Referrals and Continuation References

When a DSA holds none of the information required to satisfy a request, it can return a ***referr a l*** to the X.500 API. A referral contains the presentation addresses of one or more other DSAs that might be able to satisfy the request.

When a DSA holds some, but not all, of the information required to satisfy a request it returns a continuation reference. A continuation reference describes how the operation can be continued at one or more other DSAs. A single continuation reference returned as the entire response to an operation, with no partial results, is a referral. One or more continuation references can be included in a Partial-Outcome-Qualifier returned by a List or Search operation; see Section 1.4.8 for more information about how a Partial-Outcome-Qualifier is used. See *VSI X.500 Directory Service Programming Reference* for details of the information contained in a continuation reference.

The X.500 API passes referrals on to your application.

### 1.4.4.1. Handling Continuation References or Referrals

---

**Note**

This version of the X.500 API does not provide the Automatic Continuation service. Regardless of the setting of the Automatic-Continuation parameter, your application must handle all referrals.

---

The Automatic-Continuation parameter controls whether referrals, including continuation references, are handled by the X.500 API, or by your application.

Figure 1.6 shows a referral handled by the X.500 API. Figure 1.7 shows a referral handled by an application.

**Figure 1.6. A Referral Handled by the X.500 API**

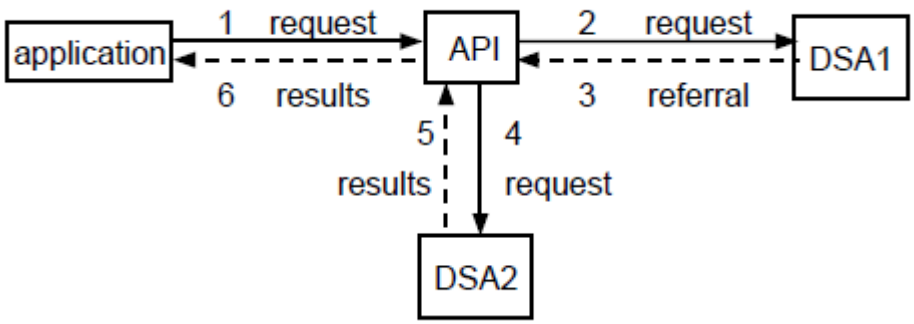
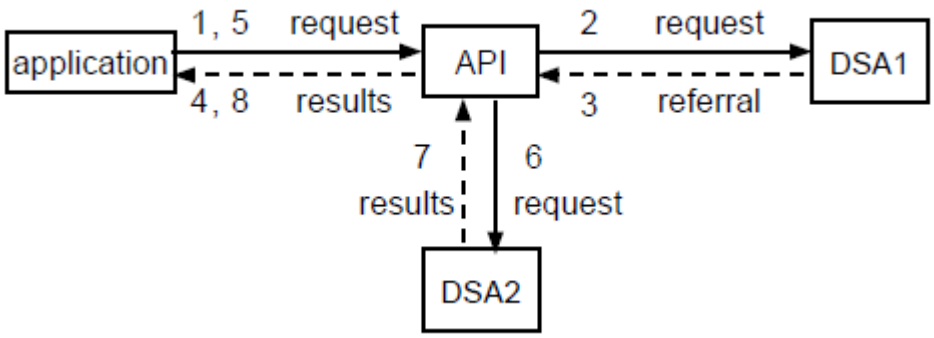


**Figure 1.7. A Referral Handled by the Application**




If the Automatic-Continuation parameter is set to *true*, the X.500 API processes all referrals and returns the final results to the application whenever practical. Use this default setting if the application does not need to deal with referrals, and you want to make the handling of referrals as simple as possible. However, the X.500 API might still return a referral to the application, for example, if it cannot contact the appropriate DSA. You need to plan what to do if this happens.

If the Automatic-Continuation parameter is set to *false* , your application must handle all referrals. Use this value if the application has special requirements that mean it is preferable for the application to deal with referrals; for example, if the user does not always want to call the function again when a Partial-Outcome-Qualifier is returned or does not want to follow a referral.

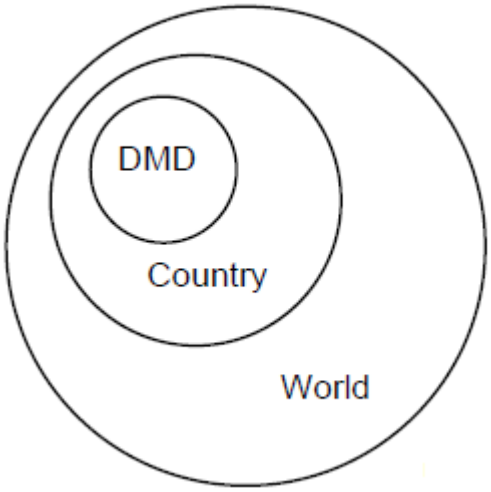
## 1.4.4.2. Referral Scope

### Note

This version of the X.500 Directory Service does not honor the setting of the Scope-of-Referral service control but assumes that it is set to *World* .

The Scope-Of-Referral parameter limits the DSAs to which the X.500 Directory Service can refer you. Figure 1–9 shows the three possible scopes.

**Figure 1.8. Scope for Returning Referrals**




The Scope-Of-Referral parameter has one of the following values:

- *DMD*

  This indicates that you want referrals only to DSAs within the same directory management domain (DMD) as the DSA in which the request originates. Typically, a DMD would be an organization, so this value would ensure that you only receive referrals to other DSAs within your organization. Therefore, you specify this option to gain access to information held within your organization; for example, a search for all meeting rooms within a department that are available at a specified time.

- *Country*

  This indicates that you want referrals to DSAs within the same country as the DSA to which you originally sent the request, including your DMD. Specify this option to gain access to information held nationally but outside your DMD. This could be information held by another organization in your country. For example, a request for the telephone number of an organization.

  The country of a DSA is determined by the value of the CountryName attribute of the entry in the DIT that represents the DSA. The CountryName of a DSA identifies the country in which the DSA is located or with which it is associated. Therefore, a DSA located in one country could have a different country as the value of its CountryName attribute, for example, a DSA belonging to an international organization that has one country as its base.

- *World*

  This indicates that you want referrals to any DSA in any DMD in any country in the world, including your country. Specify this option to gain access to information held outside of your country. This could be information held by an organization that is based in another country, for example, the facsimile number of an organization in a different country.

## 1.4.5. Using Alias Entries

An alias entry provides an alternative name for another entry and contains the Distinguished Name of that entry. This allows a single real-world object to be known by different names within the Directory, enabling different users to use the names familiar to them. See *DEC X.500 Directory Service Management* for more information about alias entries.

To discover the Distinguished Name of the entry to which an alias refers, the alias is interrogated by the Directory Service. The X.500 Directory Service uses the Distinguished Name to locate the entry to which the alias points and continues the operation with that entry. This is known as ***dereferencing*** an alias.

Use the Dont-Dereference-Alia ses service control to specify whether or not aliases that identify the target entry of operations are to be dereferenced. This is set as follows:

- To dereference aliases, use the default value of *false* for Dont-Dereference-Aliases. Use this value if you need to access the actual entry and are not interested in the alias except as a pointer; for example, if you need to modify or interrogate the entry.

- To stop aliases being dereferenced, set Dont-Dereference-Alia ses to *true*.

  Use this value if you need to perform operations on alias entries, for example, if you need to access the information held in an alias entry, or if you want to delete the alias but not the actual entry.

This service control does not apply to Add, Modify and Delete operations. The Search_Aliases parameter of the Search function further controls whether aliases are dereferenced in searches; see Section 2.5.4.5 for information about the how aliases are handled in Search operations.

## 1.4.6. Using Copy Entries

The X.500 Directory Service does not guarantee to keep copy entries consistent with each other, or with the original entry. Therefore, if you use copy entries, then you must be prepared to receive out-of-d ate data, and it must be dealt with in an appropriate way by your application.

Allow copies to be accessed if one of the following applies:

- You know the data you are accessing is stable.
- The accuracy of the information is not very important.

Do not allow copies to be used if the information you require must be accurate and up to date, and you know that the original entry is the most accurate and up to date version.

Use the Dont-Use-Copy parameter to specify whether or not requests must be satisfied by accessing original entries, as follows:

- If it is not important that the request is satisfied by accessing entries or copies of them, use the default value of *false* for Dont-Use-Copy.
- If the request must only be satisfied by accessing the original entry, set Dont-Use-Copy to *true*.

You need know the frequency with which the DSA updates its copy entries before you decide whether to use copy entries. Note that modification requests always use the original entry; copy entries are only useful for interrogations.

## 1.4.7. Setting Priorities for Operations

VSI X.500 Directory Service does not currently recognize the priority settings. All requests sent to a Digital DSA are treated with equal priority.

The Priority parameter specifies the level of priority assigned to a request when it is sent to the Directory. This priority is relative to other requests and the possible values are:

*Low*

- *Medium*
- *High*

The default value for the Priority parameter is *Medium* .

This service is not guaranteed because there is no Directory-wide system to do this. Therefore, the priority that your request receives depends upon the DSA that the request is sent to. To use the Priority parameter, you need to understand how different implementations deal with the priority settings.

## 1.4.8. Specifying the Operation Start State

When you call an X.500 API function, it invokes a directory operation. It is possible that a previous request to perform the operation returned partial results in a Partial-Outcome-Qualifier; therefore, you need to invoke the operation again to obtain further results. The X.500 API provides the means of specifying whether the request is already partly processed. Use the Operation- Progress parameter to specify the state the operation has reached. The possible values are:

- *Operation-Not-Started*, no work has been done

- *Proceeding*, some work has been done
- *Completed*, all the work has been done

The default value is the constant DS_OPERATION_NOT_STARTED.

This parameter normally takes the default value but there are circumstances in which you would choose to use another value, for example, to deal with operations that return a Partial-Outcome-Qualifier.

### 1.4.9. Using Default Values

If you want to use the system default values for all the parameters in an OM Context object, you can supply the constant DS_DEFAULT_CONTEXT instead of an object.

You can also use the information in the DUA defaults file to override the system default values. When you create an OM Context object, the object is initialized with the values from the DUA defaults file, or with the system default values if a value is not specified in the DUA defaults file.

Similarly, you can specify the constant DS_DEFAULT_S ESSION in place of an OM Session object, or you can use the information in the DUA defaults file to control directory session characteristics. See Section 2.1.2 for more information.

## 1.5. Interworking

This section explains the planning you must do to ensure that your application can interwork with non-Digital implementations of the X.500 Directory Service. This interworking is essential if your application needs to contact another vendor's DSA through VSI API.

You need to understand fully the characteristics of the particular DSA that your application interacts with. The CCITT X.500 Series of Recommendations includes many features that are implementation-defined, such as how certain service controls are implemented. Other features are defined as optional, and therefore might not be supported by a particular DSA.

You also need to understand the characteristics of VSI X.500 API. For example, you must be aware of the Digital extensions to the standard X.500 API, as other vendors' DSAs do not necessarily support the same extensions.

Similarly, you need to be aware of what optional functionality, as specified in the standard X.500 API, is not supported by VSI X.500 API. See Appendix A for comprehensive lists of Digital extensions to the X.500 API and functionality denoted as optional in Version 1 of the X/Open CAE Specification *API to Directory Services (XDS)* but not supported by VSI X.500 API.

You need to consider how X.500 Directory Service implementations differ and to allow for that difference. You also need to plan the use of functionality that is optional or an extension, and how to work around it if required.

## 1.6. Speed and Accuracy

This section explains the planning required to build an application that gives an appropriate response time and level of accuracy. Accuracy means complete, up to date results.

You need to decide what response time is required, compared with the need for accuracy. If your application is used interactively, then a rapid response time is important, and you might decide to place a

time limit on all operations. This would mean that a user would never have to wait longer than the time limit, but it could also mean that the results of the operation might be incomplete, or empty.

For interrogation operations, it is faster to access a locally held copy of an entry, rather than the actual entry held remotely. However, the information stored in a copy is not necessarily up-to-date or correct. See Section 1.4.6 for further information on entries and copies.

Since the X.500 Directory Service can store many objects across any number of DSAs, some operations, such as searches, could take a long time to complete. An application must be prepared to handle long delays in an appropriate way.

Plan for long delays so that operations that return quickly can be processed while waiting for those that take a long time. If possible, users should be warned of any delay so that they can determine whether they have the time to do something else while they wait. If arbitrary delays are unacceptable, place a time limit on the operation.

# 1.7. Security

You need to consider the following types of security:

- Access control (ensuring that users cannot gain access to data or applications for which they do not have access rights)
- Authentication (ensuring that users are valid users who are allowed to use the directory)

You must not rely on the integrity of unencrypted information held in the Directory to provide security for sensitive information. For example, an application should not provide elevated privileges to a user based upon information stored in the Directory. If information important to security decisions is stored in the Directory, that information must be independently verified by the application.

The CCITT X.500 Series of Recommendations does not define any particular security policies. Implementations differ in the methods they use to provide security, and the level of security they provide. Therefore, you need to be aware of the security policies at any DSA that your application accesses. See *DEC X.500 Directory Service Management* for information about security features provided in VSI X.500 Directory Service.

## 1.7.1. Access Control

VSI X.500 API provides security based upon access controls defined for portions of the DIT. Access controls are used to prevent users accessing data or applications that they are not authorized to access.

Groups of users with particular access rights are identified by their X.500 Distinguished Names.

In response to Read operations, your application must be able to handle responses that contain information that is incomplete due to the access controls on the data. On other operations, it must be able to handle errors caused by access controls.

## 1.7.2. Authentication

You need to consider two types of authentication:

- Authentication by the directory service to permit access to data held in the directory. This is done as part of the Bind operation. The VSI X.500 Information Management utility, DXIM, is an example of an application that does this. See the DXIM on-line help for information about authenticated binds.

- Application authentication, where the application verifies the identity of a user by comparing a password supplied with a password held in the directory. The user is then granted certain privileges or access rights. Not all applications need to implement this type of authentication.

See *DEC X.500 Directory Service Management* for information about authentication.

# 1.8. The User Interface

This section explains what you need to understand or consider when planning the user interface of your application. Some attribute values are stored in ASN.1, so your application needs to convert the information to a format suitable for display. See Section 2.5.5 for information about how to handle the data returned from the Directory.

X.500 names can be long and complicated. You need to consider how you can present them to the user of your application in a simpler and more usable way.

Wherever possible, you should provide users with easy-to-use alternatives to distinguished names. The alternatives that are available include:

- Using default values for function argument values that are used most. For example, if the application is usually concerned with entries within an organization or department, use default values for the first RDNs of Distinguished Names, including the OrganizationName or an OrganizationalUnitName.
- Using logical names that are translated into global names by the application; for example, a mnemonic that points to a particular file, or invokes an application.
- Using nicknames that are translated into global names by the application, for example, a nickname that translates to the Distinguished Name of a department's printer.

If you use defaulting or translation of names, then to ensure that the correct name is entered, you must give the user the following options:

- To enter a fully typed global name
- To see the full name resulting from any defaulting or translation

In planning the format of the screens for the user interface, you need to consider how much space is required to hold all the information supplied by the application and the user. Global names are typically much longer than local names, therefore sufficient space needs to be allocated for them.

# 1.9. XDS Example Programs

The X.500 API component of the VSI X.500 Directory Service includes a set of callable routines that illustrates how to use the XDS and OM routines. You can also use these routines, which are known as the XDSHLI routines, in your application.

The X.500 API also includes simple example programs that read and search the directory.

The file SYS$EXAMPLE S:[DXD]XDSHLI.H contains details of the XDSHLI routines, the example programss, and related files.

# Chapter 2. Application Task Descriptions

This chapter describes each of the tasks that you might want an application to do. It explains the services required to perform the tasks. It also provides information about selecting and using the X.500 API functions.

Section 1.9 contains information about the X.500 Directory Service high-level interface, XDSHLI, which is a set of routines that perform common interface tasks using the X.500 API. You can use these routines as examples of how you can perform similar tasks in your application, or you can call the XDSHLI routines themselves.

## 2.1. Connecting and Disconnecting

Before any information can pass between the application and the X.500 API, the application must initialize the API, and the API must also establish a connection between itself and the Directory. Having exchanged information with the Directory, the application must signal that it has finished and must close down the connection. These steps are explained, in the order that they should be performed, in Section 2.1.1 to Section 2.1.6.

### 2.1.1. Initializing the Interface

The first task of an application is to initialize the X.500 API by calling the Initialize function. This function must be called before any other X.500 API functions are called.

The Initialize function establishes the workspace that your application uses during its communication with the Directory. This workspace supports the standard Directory Service Package (the xds.h header file). Therefore, it supports the objects described in *VSI X.500 Directory Service Programming Reference*. It does not support other packages or extensions that affect the behavior of those applications that use only the standard features. For details of negotiating other features of the workspace, see Section 2.1.4.

The Initialize function has no arguments.

### 2.1.2. Opening a Directory Session

Call the Bind function to open a session with the Directory and to specify the Directory System Agent (DSA) that you want your application to communicate with. The Bind function has two arguments: Session and Workspace:

- For Session, supply either a Session OM object, which can contain the name and address of the DSA to bind to, or the constant DS_DEFAULT_ SESSION. If you provide an object that does not contain details of the the DSA to bind to, the DUA.KnownDSAs.paddr and DUA.KnownDSAs.ae_title values from the DUA defaults file are used. If you specify the constant, the values from the DUA defaults file are used. See *DEC X.500 Directory Service Management* for information about the DUA defaults file.

- For Workspace, specify the workspace to be associated with the session, as returned by the Initialize function.

An application must use the Bind function before it can use any of the interrogation or modification functions. Therefore, the Bind function must be called at least once, after the Initialize function but before any other function calls.

It is possible for an application to call the Bind function many times between calling the Initialize function and calling the Shutdown function. The interface supports multiple concurrent sessions within the same Directory communication. This allows you to:

- Implement an application as a single process, but enable it to interact with the Directory using several identities. Do this by specifying the same Session object as the argument to several Bind function calls. This invokes many sessions with the same DSA.
- Invoke an application that interacts directly and concurrently with different parts of the Directory. Do this by specifying different Session objects for each Bind function call. This enables you to communicate directly with many DSAs.

You are bound to a DSA and therefore able to communicate with it, until you pass the Session object that identifies that DSA as the argument to the Unbind function. See Section 2.1.5 for details of closing a Directory session.

Note that the VSI implementation will attempt to rebind to the DSA if the DSA becomes unavailable once bind has been called. If the DSA cannot be contacted, an error is returned.

## 2.1.3. Setting the Default Service Controls

Most of the X.500 API routines have a Context argument. The Context object contains settings of the service controls that apply to the Directory operation. You can use the DUA defaults file to set default values for service controls. See *DEC X.500 Directory Service Management* for more information about the DUA defaults file.

## 2.1.4. Negotiating the Version of Interface and Service

Call the Version function to negotiate certain features of the workspace returned by the Initialize function. Negotiate these features after the interface has been initialized. The only negotiable feature of the workspace is support of objects defined in the Basic Directory Contents Package, the Strong Authentication Package and the MHS Directory User Package respectively.

See Section 3.2.1, and the *DEC X.500 Directory Service Programming Reference* for details of these packages.

The Version function has two arguments: Feature-list and Workspace. Feature-list is a list of object identifiers. Each object identifier represents a feature of the workspace, and is shown in the following list:

- The xdsbdcp.h header file is identified by the object identifier DS_BASIC_DIR_CONTENTS_PKG.
- The xdssap.h header file is identified by the object identifier DS_STRONG_AUTHENT_PKG.
- The xdsmdup.h header file is identified by the object identifier DS_MHS_DIRECTORY_USER_PKG.

For details of what the header files contain, see the *DEC X.500 Directory Service Programming Reference*. Note that the VSI implementation does not support these.

The Workspace argument specifies the workspace for which the features are to be negotiated. See the description of the Bind function in *DEC X.500 Directory Service Programming Reference* for information about other negotiable features.

### 2.1.5. Closing a Directory Session

Call the Unbind function to end a session with the Directory Service. This function has one argument, the Session object identifying the session that you want closed.

If any asynchronous operations were initiated during the session, you must first ensure that they all have their results returned by the Receive-Result function, or are terminated by the Abandon function before you close the session.

A Session object that identifies a closed session cannot be used as the argument to any function except Bind. You can use the Session object of a closed session as the argument to the Bind function to reopen the session.

### 2.1.6. Releasing a Workspace

Call the Shutdown function to shut down a workspace previously set up by the Initialize function. This allows the Directory Service to release resources such as memory. After calling this function, an application cannot call any other interface functions until it has called the Initialize function to establish a workspace. The Shutdown function also deletes any service-generated OM objects, so you must ensure that they are no longer required by your application before calling the function.

The Shutdown function has one argument, workspace, which specifies the workspace that is to be deleted.

You should ensure that all open sessions are closed by calling the Unbind function before you call the Shutdown function to ensure that all resources are released.

### 2.1.7. Using Other DSAs

The X.500 API supports multiple concurrent sessions; therefore, you can be bound to, and communicate with, many DSAs concurrently. See Section 2.1.2 for details of binding to a DSA.

The first parameter of most API functions, except Initialize, Version and Shutdown, is a Session object that identifies the DSA that is to be used to service the request. Therefore, you can use different DSAs for any individual operations by passing different Session objects to the function that invokes the operation.

You can only pass a Session object that identifies a DSA to which you are bound. Therefore, you need to bind to the DSA that you want to use (the target DSA) if you have not already done so, or if you have bound to it and have subsequently closed the session by calling the Unbind function.

## 2.2. Creating and Manipulating Objects

The application can use the workspace returned by the Initialize function to create and manipulate private and public Object Management (OM) objects. See Section 2.1.1 for details of the Initialize function. The application then uses these objects as arguments to X.500 API functions. For example, the application can create an object of OM class Name in the workspace and then use that object as an argument of a Read function call.

To create a private object:

- Use the OM Create function to create the object; for example, to create an object of the OM class Context.
- Use the OM Put function to place the required values in the object; for example, to set the attributes Size-Limit and Time-Limit in the object of OM class Context that you have just created.

You can create a public object in one of the following ways:

- To create a client-generated public object to store static data, declare the necessary data structure or data structures for the object that you want to create; for example, a descriptor list that represents an object of OM class DS_DN (a Distinguished Name). You initialize values in the public object as part of the declaration. There are macros available to help you create a public object. See *OSI-Abstract-Data Manipulation* and the example programs described in Section 1.9 for details.
- To create a client-generated public object to store dynamic data, allocate the storage dynamically, and then assign values to the fields in the descriptor list.
- To create a service-generated public object that is a copy of a private object, use the OM Get function. This allows you to read attribute values of an object, for example, to examine the values passed as an object of the OM class Context.

  Public objects created in this way can be read but not modified by the application.

You can manipulate private objects as follows:

- Add attributes to the object using the OM Put function. These attributes can be either simple data values or objects.
- Access attribute values using the OM Get function.
- Determine whether an object is of a specified object class, or any of its subclasses, using the OM Instance function.
- Delete attributes using the OM Remove function.
- Delete the object using the OM Delete function.
- Make a copy of the object using the OM Copy function.

The OM Delete and Instance functions can also be applied to service-generated public objects. No other OM functions can be applied to public objects.

See *OSI-Abstract-Data Manipulation* for details of using the OM functions.

# 2.3. Using Asynchronous Operations

This section describes the tasks connected with asynchronous operations.

The directory interface allows applications to use all the interrogation and modification functions synchronously or asynchronously. The interface provides access to two additional functions for asynchronous operations:

- The Receive-Result function allows you to receive the result of an asynchronous operation.
- The Abandon function allows you to abandon the result of a pending asynchronous operation.

To invoke an asynchronous operation, you must call the corresponding X.500 API function with the Asynchronous parameter in the Context argument set to *true*. The Directory Service then invokes the asynchronous operation, unless an error occurs that prevents it from invoking the operation.

When the Directory Service invokes an asynchronous operation, it returns an Invoke-ID to the X.500 API. An Invoke-ID uniquely identifies the operation among all other outstanding asynchronous operations invoked in a particular session. This enables you to abandon the operation, or to identify the results of the operation when results are returned using the Receive-Result function (see the Section 2.3.1).

When an application has invoked an asynchronous operation, it must call the Receive-Result function to determine whether the Directory has returned the result of that operation.

You can improve the performance of your application on an ULTRIX™ or DEC OSF/1 system by not calling the Receive-Result function until a response has been returned by the Directory.

An application can use the ULTRIX or DEC OSF/1 Select system call to determine when to call Receive-Result, as follows:

1. Obtain the value of the DS_FILE_DESCRIPTOR attribute from the Bound Session Object returned when the session was created. (The application must not attempt to write to this file descriptor.)
2. Call Select, supplying this file descriptor as a parameter. The return status indicates whether a response has been returned by the Directory.
3. Call Receive-Result to collect the data ready to be processed. Note, however, that the application should not assume that Receive-Result will return the result of the operation. If the size of the result is large, you may have to call the Receive-Result function more than once to collect all the data.

Use of the Select service is optional on ULTRIX and DEC OSF/1 systems; you can collect the results of asynchronous operations by calling Receive-Result without first calling Select. There is no equivalent service available to applications on OpenVMS systems.

If you have an asynchronous operations outstanding and you call a synchronous operation, the return status of the Select call may incorrectly indicate that no data has been returned. You must always call the Receive-Result function after each synchronous operation if you have any outstanding asynchronous operations.

## 2.3.1. Receiving Results of Asynchronous Operations

Call the Receive-Result function to access the results of an outstanding asynchronous operation.

When you call the Receive-Result function, you must specify the session in whose outstanding operations you are interested. This indicates the DSA to which the requests were originally sent. You specify this as the Session parameter of the Receive-Result function.

The Receive-Result function can return results from any outstanding asynchronous operation. Therefore, to identify the asynchronous operation to which these results relate, it also returns the Invoke-ID of that operation.

A call to the Receive-Result function returns four values, as follows:

- Status, which reports that the Receive-Result operation completed successfully.
- Completion-Flag, which reports whether there were any completed outstanding operations.
- Operation-Status, which reports whether the asynchronous operation completed successfully.
- Result, which contains any results from the asynchronous operation.

If at least one outstanding asynchronous operation has completed, the following are also returned:

- The Completion-Flag parameter value of DS_COMPLETED_OPERATION.
- The Invoke-ID of the asynchronous operation whose results have been returned.
- The Operation-Status parameter value of DS_SUCCESS, if the asynchronous operation completed successfully.
- Any results from the asynchronous operation in the Result parameter, if the asynchronous operation completed successfully. If the operation is one that does not return results (Add-Entry, Modify-Entry, Modify-RDN, or Remove-Entry), then the Result parameter value is DS_NULL_RESULT.
- If the asynchronous operation did not complete successfully, the Operation-Status parameter returns an error value that indicates the error that occurred during the execution of the operation, and the result parameter is unspecified.

If there are outstanding asynchronous operations, but none have completed, the Completion-Flag parameter value of DS_OUTSTANDING_OPERATIONS is returned.

If there are no outstanding operations, the Completion-Flag parameter value of DS_NO_OUTSTANDING_OPERATION is returned.

Any results returned in the Result parameter are in the form of an OM object, and of a class appropriate to the results. This is a private object; therefore, you need to use the OM functions to determine its class and contents.

## 2.3.2. Abandoning Asynchronous Operations

Call the Abandon function to tell the Directory Service that the application is no longer interested in the results of an operation. The results can never be accessed once they have been abandoned, even if the Receive-Result function is called.

The Abandon function applies only to interrogations. The interface does not allow applications to abandon modifications.

However, calling the Abandon function does not guarantee that an abandon operation is requested of the X.500 Directory Service. If an abandon operation is invoked, the X.500 Directory Service does not necessarily abandon the outstanding asynchronous operation itself. How a DSA behaves when it receives an abandon request is implementation-defined.

Specify the session as the Session parameter of the Abandon function that was the target of the operation to be abandoned. Specify the Invoke-ID of the operation that you want to abandon as the Invoke-ID parameter of the Abandon function.

Note that even if the Abandon function returns an error, the result of the asynchronous operation is not returned.

### 2.3.3. Use of Asynchronous Operations

Asynchronous operations are ideal for applications which need to continue to work while waiting on the results of a previously issued operation. An example of this is the Motif version of the DXIM utility, which is a single-threaded windowing application in which other parts of the application can still be used while waiting for results of an operation.

## 2.4. Continuing Operations

When an operation returns partial results to the X.500 API it is possible that these results are sufficient for your needs, or you might need further results. If a function call returns partial results and you need further results, then you must call the function again.

The information required to continue the operation is foun d in the object of OM class Continuation-Ref in the Partial-Outcome-Qualifier that was returned by the incomplete operation.

When you call the function again, you specify the same parameter values except for the following:

- The Operations-Progress parameter of the Context object has the same value as the Operations-Progress attribute of the Continuation-Ref. This value should be *Proceeding*.
- The Session parameter specifies the name and presentation address of a DSA returned in the Access-Point s attribute of the Continuation-Ref. You must bind to this DSA before you call the function.

## 2.5. Interrogating the Directory Information Base

You can make as many interrogations as you require during a directory session. The interface provides four ways of interrogating the Directory to retrieve information:

- Read attributes of an entry.
- List the immediate subordinate entries of a specified entry.
- Compare attributes of an entry with specified values.
- Search for an entry or entries.

The information returned in response to a request depends on the the request itself, and on the access control that applies to entries accessed to satisfy the request. Refer to Section 1.7.2 for information about access controls.

### 2.5.1. Reading Entry Information

Call the Read function to look at a specified entry. You can specify the combination of the entry's attributes and attribute values that you want to see.

See Section 1.9 for information about an example program that reads information from the directory.

#### 2.5.1.1. Specifying the Entry to Read

You must specify the entry from which you want to obtain the information. You specify the entry by entering its name as the Name parameter of the Read function.

### 2.5.1.2. Specifying the Information Returned

Specify the information you want returned by passing an object of OM class Entry-Info-Selection in the Selection parameter of the Read function. The attributes of this object are as follows:

- All-Attributes. This is a Boolean value.
- Attributes-Selected. This is a list of zero or more attribute types.
- Info-Type. This has one of the following values:
  - *types-only*
  - *types-and-values*

You can also pass one of the following constants, in place of the OM Entry-Info- Selection object:

- DS_SELECT_NO_ATTRIBUTES
- DS_SELECT_ALL_TYPES
- DS_SELECT_ALL_TYPES_AND_VALUES

Some examples of read operations that you can perform, and the required information in the Entry-Info-Selection object, are:

- If you want to return the Distinguished Name of an entry to confirm the existence of that entry, then:
  - Set All-Attributes to *false* .
  - Specify an empty list as Attributes-Selected or pass DS_SELECT_NO_ ATTRIBUTES.
- If you want to return all the attribute types in an entry to determine what kind of information is stored in that entry, then:
  - Set All-Attributes to *true*.
  - Set Info-Type to *types-only* or pass DS_SELECT_ALL_TYPES.
- If you want to return all the attribute types and values in an entry to obtain all the known information about that entry, then:
  - Set All-Attributes to *true*.
  - Set Info-Type to *types-and-values* or pass DS_SELECT_ALL_TYPES_ AND_VALUES.
- If you want to discover whether an entry has attributes of certain types, do the following:
  - Set All-Attributes to *false* .
  - Specify the attributes about which you want information returned in Attributes-Selected.
  - Set Info-Type to *types-only* .
- If you want to discover the types and values of some of the attributes in an entry, for example, to return a person's address and telephone number, then:
  - Set All-Attributes to *false* .

- Specify the attributes about which you want information returned in Attributes-Selected.
- Set Info-Type to *types-and-values* .

### 2.5.1.3. Obtaining Results

The results of a Read operation are returned as an object of the OM class Read-Result. This object contains the following information:

- The Distinguished Name of the target entry specified in the Name parameter.
- A Boolean flag indicating whether the information came from the actual entry or a copy of the entry. The flag is set to *false* if the information came from a copy entry and *true* if the information came from the original entry. If the Dont-Use-Copy service control was in operation for the request, this flag is set to *true* because the information came from the actual entry. See Section 1.4.6for details of this parameter.
- Any requested attribute types and values.

The Read-Result object is passed as the Result parameter of the Read function.

## 2.5.2. Listing an Entry

Call the List function to list the RDNs of all the immediate subordinate entries of a specified entry. For example, if you are developing a directory browsing utility, then you could use the List function to allow the user to look for entries one level down in the DIT. This allows the user to find an entry when they do not know the Distinguished Name but can locate the area of the Directory where the entry exists. The List function returns only the RDN of an entry and not the full Distinguished Name as is the case with the Search function (see Section 2.5.4).

### 2.5.2.1. Specifying What to List

Specify, in the Name parameter of the List function, the entry whose immediate subordinates you want to list.

### 2.5.2.2. Obtaining Results

The results of a List operation are returned as an object of the OM class List-Result. This contains the following information:

- The RDN of each subordinate entry, with Boolean values indicating whether the entry is an alias and whether the information was obtained from the original entry or a copy.
- The Distinguished Name of the target entry specified in the Name parameter, if an alias was dereferenced to find it.
- If the result is incomplete, an object of OM class Partial-Outcome-Qualifier indicating this and why it has not completed.

  If a Partial-Outcome-Qualifier is returned, this may contain a referral or a continuation reference. See Section 1.4.4 for details of these.

The List-Result object is passed as the Result parameter of the List function or as part of the Receive-Result function.

## 2.5.3. Comparing Attribute Values and Specified Values

The Compare function allows you to specify an entry and see whether one of its attributes has a specified attribute type and value. The Directory Service responds to this function by returning *true* or *false* .

Use this function to verify the value of an attribute, and to confirm that an entry has an attribute of a given type with the specified value. This is particularly useful to verify the value of attributes that contain information that is protected by access controls. For example, you could use the Compare function to verify user passwords. This function does not return the attribute value, so you can only use it to verify the password provided.

### 2.5.3.1. Specifying the Entry to Compare

You must specify the entry that you want to compare with the actual value or values. You do this by specifying its name as the Name parameter of the Compare function.

### 2.5.3.2. Specifying Values to Compare

You must specify, as an object of OM class AVA, the attribute value assertion (AVA) that you want compared with the specified entry. The AVA object is passed as the AVA parameter of the Compare function. This contains the attribute type and value that you want to compare with the entry.

The attributes in the entry and in the AVA are compared in two steps:

- The AVA attribute type is compared with the attribute types in the entry to see whether there are any that match.
- If the types match, the attribute value in the AVA is compared with those in the attribute in the entry of the corresponding type to see whether they are equal.

  Equality is based on the matching rules attached to the attribute syntax of the attribute types that are being compared. For example, if the syntax is integer, the numeric values must be the same. See the *DEC X.500 Directory Service Management* for information about attribute types, attribute syntaxes, and matching rules.

If the types and values are equal, then the attribute and AVA are equal.

### 2.5.3.3. Obtaining Results

The results of a Compare function are returned as an object of OM class Compare-Result. This contains the following information:

- The Distinguished Name of the target entry specified in the Name parameter, if an alias was dereferenced to find it
- A flag indicating whether any of the values matched
- A flag indicating whether the comparison was made against the actual entry or a copy of it

The Compare-Result object is returned as the Result parameter of the Compare function.

## 2.5.4. Searching for an Entry

Call the Search function to search for entries of interest. You must specify an area of the DIT in which to search for any entries that have attributes and attribute values that match a filter. The filter is evaluated

against every entry within the specified area. You can also specify the information that you want returned about entries that match the filter.

See Section 1.9 for information about the example programs supplied with the Digital X.500 API. The file XDS_SEARCH_EXAMPLE_1.C (OpenVMS) or xds_search_example_1.c (ULTRIX and DEC OSF/1) contains an example of a program that searches for information in the directory.

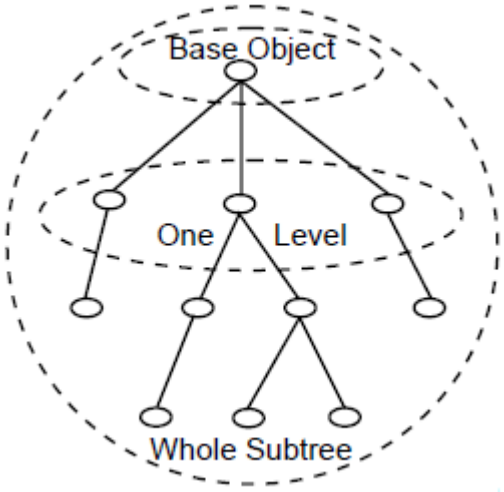
### 2.5.4.1. Specifying the Starting Point of the Search

Specify the entry from which the search must start. You do this by specifying its name as the Name parameter of the Search function.

### 2.5.4.2. Specifying the Depth of the Search

You must specify how much of the Directory you want to search. You can choose one of three depths of search, as shown in Figure 2.1.

**Figure 2.1. Depths of Search**




Specify the depth of the search by passing one of the following three constants as the value of the Subset parameter:

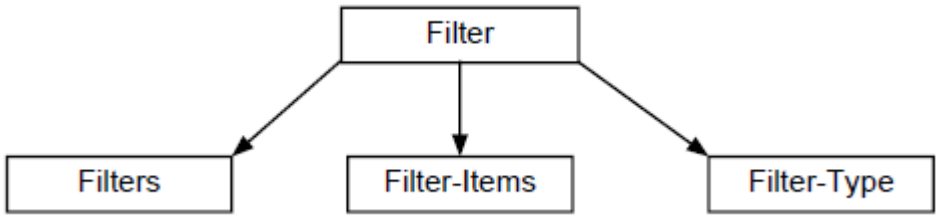
- DS_BASE_OBJ ECT. Use this value to search only the target entry that you specified in the Name parameter.
- DS_ONE_LEVEL. Use this value to search the immediate subordinate entries of the specified target entry.
- DS_WHOLE_SUBTREE. Use this value to search the specified target entry and all its subordinate entries.

### 2.5.4.3. Omitting Entries from the Search

You can use the Filter parameter specify the entry you are looking for and reduce the size of the search results. For example, when you are searching for certain employees within an organizational unit, you only want information returned about Organizational Person entries. All other classes of entry should be omitted from the search result.

To specify the requirements that an entry must satisfy to be included in the search, pass an object of the OM class Filter as the Filter parameter of the Search function. This object has the attributes shown in Figure 2.2.

**Figure 2.2. Attributes of the OM Class Filter**



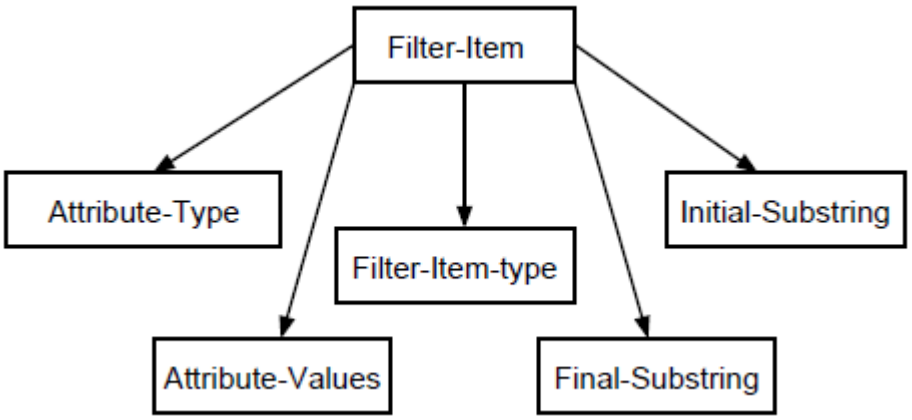

If no filter is specified for a search, all entries are returned, subject to access control and DSA service controls.

Specify a Boolean operator as the attribute Filter-Type. Filter-Type joins together the elements of the Filter to determine whether or not an entry satisfies it. The elements of the filter are the Filter-Items and nested Filters, each of which makes an assertion about the attributes of an entry. Decide which Boolean operator to use as follows:

- If you want the Filter to succeed only if all the assertions succeed, then use the value *and* .
- If you want the Filter to succeed if any of the assertions succeed, then use the value *or*.
- If you have a Filter with one nested Filter, or Filter-item, and you want the Filter to succeed if the nested Filter or Filter-item fails, then use the value *not*.
- If you have a Filter with one nested Filter-item, and you want the Filter to succeed if the nested Filter-item succeeds, then use the value *item* .

Specify a list of assertions about the existence or values of an entry attribute type as the Filter-Items attribute. Filter-Items contains any number of objects of the OM class Filter-Item. The attributes of the OM class Filter-Item are shown in Figure 2.3.

**Figure 2.3. Attributes of the OM Class Filter-Item**




Specify each assertion you want to make as a Filter-Item by declaring values for its attributes, as follows:

- Attribute-Type

- Attribute-Values
- Filter-Item-Type
- Initial-Substring
- Final-Substring

Specify the attribute type that an entry must contain as the Attribute-Type attribute. Specify the attribute value of the attribute type that an entry must contain as the Attribute-Values attribute. The Attribute-Values attribute must have exactly one value except when you specify a Filter-Item-Type of *present* or *substrings* ; see the description of these values in the following list for details.

Set the value for Filter-Item-Type as follows:

- If you want to test whether the entry contains an attribute of the specified type that has at least one value that is approximately equal to the value specified, then use the value *approximate-match* .
- If you want to test whether the entry contains an attribute of the specified type that has at least one value that is equal to the value specified, then use the value *equality* .
- If you want to test whether the entry contains an attribute of the specified type that has at least one value that is greater than, or equal to, the value specified, then use the value *greater-or-equ al* .
- If you want to test whether the entry contains an attribute of the specified type that has at least one value that is less than, or equal to, the value specified, then use the value *less-or-equ al* .
- If you want to test whether the entry contains an attribute of the specified type regardless of its value, then use the value *present*.

  The Attribute-Values attribute is ignored when you specify this value.
- If you want to test whether the entry contains an attribute of the specified type that has a value containing all the specified substrings in the given order, then use the value *substring* .

  The Attribute-Values attribute can contain zero or more values; each value contains one substring.

If you want to test whether the last part of an attribute value is a particular substring, then declare that substring as the value of Final-Substring.

If you want to test whether the first part of an attribute value is a particular substring, then declare that substring as the value of Initial-Substring.

Note that the above values must not overlap. The following example shows an object of the OM class Filter, filter, whose purpose is to locate the entry that represents Joan Smith who works in the Sales department.

```
#define Astr1 DS_A_COMMON_NAME
#define Astr2 DS_A_ORG_UNIT_NAME
#define str1 "Joan Smith"
#define str2 "sales"

static OM_descriptor
filter_item_1 [] = {
OM_OID_DESC(OM_CLASS, DS_C_FILTER_ITEM),
```

```
{DS_ATTRIBUTE_TYPE, OM_S_OBJECT_IDENTIFIER_STRING, Astr1},
{DS_ATTRIBUTE_VALUES, OM_S_PRINTABLE_STRING, OM_STRING(str1)},
{DS_FILTER_ITEM_TYPE, OM_S_ENUMERATION, DS_APPROXIMATE_MATCH},
OM_NULL_DESCRIPTOR
},
filter_item_2 [] = {
OM_OID_DESC(OM_CLASS,DS_C_FILTER_ITEM),
{DS_ATTRIBUTE_TYPE, OM_S_OBJECT_IDENTIFIER_STRING, Astr2},
{DS_ATTRIBUTE_VALUES, OM_S_PRINTABLE_STRING, OM_STRING(str2)},
{DS_FILTER_ITEM_TYPE, OM_S_ENUMERATION, DS_APPROXIMATE_MATCH},
OM_NULL_DESCRIPTOR
},
filter [] = {
OM_OID_DESC(OM_CLASS,DS_C_FILTER),
{DS_FILTER_ITEMS, OM_S_OBJECT, {0,filter_item_1}},
{DS_FILTER_ITEMS, OM_S_OBJECT, {0,filter_item_2}},
{DS_FILTER_TYPE, OM_S_ENUMERATION, DS_AND},
OM_NULL_DESCRIPTOR
};
```

The filter object contains two subobjects of OM class Filter-Item; fil- ter_item_1 and filter_item_2. The first, filter_item_1, is only *true* if the entry has an attribute of type Common-Name, with a value of "Joan Smith".

The second, filter_item_2, is only *true* if the entry has an attribute of type Org-Unit-Name, with a value of "Sales".

The filter object has a Filter-Type value of *and* . Therefore, filter is only

*true* if both Filter-Items are *true*.

## 2.5.4.4. Specifying the Information Returned

You can use the Search function to return information about the attribute types and values of all the entries foun d by the search. Specify what information you want returned by passing an object of OM class Entry-Info-Selection in the Selection parameter of the Search function. The attributes of this object are the same as those specified for the Read function (see Section 2.5.1.2). The examples given in Section 2.5.1.2 also apply to the Search function.

## 2.5.4.5. Specifying Whether to Search Aliases

The treatment of alias entries in a search is determined by the Search_Aliases parameter of the search function and the Dont Dereference Aliases service control. The service control determines whether an alias in the base object of the search is dereferenced. The Search_Aliases parameter determines whether aliases in the subordinate entries of the base object are to be dereferenced. If they are dereferenced then the actual entry can be accessed through the information in the alias entry. Specify a Boolean value as the Search-Alia ses parameter of the Search function, as follows:

- *True* indicates that all aliases are dereferenced. This means that the search includes any entry foun d by dereferencing an alias and all its subordinate entries, as long as the search does not exceed the depth of search specified in the Subset parameter.
- *False* indicates that no aliases are dereferenced.

Refer to Section 1.4.5 for more information on the Dont-Dereference-Alia ses parameter.

### 2.5.4.6. Obtaining Results

The results of a Search operation are returned as an object of the OM class Search-Result. This object contains the following information:

- The requested information from each entry within the search depth that satisfied the Filter. This includes:
  - A Boolean value indicating whether the information was obtained from the original named entry or a copy
  - The Distinguished Name of the entry
- The Distinguished Name of the entry specified in the Name parameter, if an alias was dereferenced to find it.
- Possibly, an object of OM class Partial-Outcome-Qualifier that indicates the result was incomplete, and why it was not completed.

  If a Partial-Outcome-Qualifier is returned, it may contain a referral or a continuation reference. See Section 1.4.4 for details of these.

The Search-Result object is passed as the Result parameter of the Search function.

## 2.5.5. Using the OM-Get Routine to Extract Data From a Read-Result Object

This section describes two ways you can use the OM-Get function to extract data from a Read-Result object. It also explains the advantages and disadvantages of each method.

The Read-Result that you get back from an Abstract Service is a hierarchical data structure which contains nested structures. The OM-Get function provides features for you to obtain data from any desired level in the structure. You can either extract all the data values or extract a pointer to a sub-object.

There are two ways to access the data within the structure.

1. Use the OM-Get function to work your way down the nested structure, getting one level at a time.
2. Get all the data back at once and then work your way through the nested structure as a public object (that is, a table of OM-descriptors).

Which method you use depends on:

- The amount of data you expect back

  Extracting one level at a time (op tion 1) does not duplicate data (from the private object to a public object) until you actually get the values you want.

  Extracting all the data at once (op tion 2) will duplicate all the data that you ask for.

- Programming complexity

  It may be easier to think about one level at a time and to use a sequence of OM-Get functions to access each level until you get the values you desire. However, if you need to use some of the data

values first and then some later, you may find it better to extract all the data in one function and then have your own routines to extract the exact data you require each time.

- Whether run-time speed is more important than memory usage

  If run-time speed is of the utmost importance and memory usage is no problem, then extracting all the data at once (op tion 2) would avoid the overhead of multiple XOM service calls. That overhead should not normally be noticed unless the code is critical to the performance of the application.

You can also use a combination of these methods.

The following example code extract shows how to extract the data one level at a time (op tion 1) using the exclusions argument in the OM-Get function. The example uses the public trace object dsX_trace_object routine.

```
/* declare a OM-type-list structure and variables to hold pointers
to the entry, DS_object, and RDNS sub-object:
*/
OM_type included_types[2];
OM_public_object spub_entry;
OM_public_object spub_DS_object;
OM_public_object spub_RDNS;

/* and set up the OM attributes you want to get first: */
      included_types[0] = DS_ENTRY;
      included_types[1] = OM_NO_MORE_TYPES;
/* now get only a pointer to the first sub-object, the entry */
  om_status = om_get(read_result,
      OM_EXCLUDE_ALL_BUT_THESE_TYPES+OM_EXCLUDE_SUBOBJECTS,
           included_types, OM_FALSE, 0, OM_ALL_VALUES,
           &spub_entry, &desc_count);

/* the object spub_entry now contains only the
  OM-descriptor for an entry-information object */

dsX_trace_object(spub_entry);
/* Now, use OM-Get again to extract the distinguished name from
the entry information. */

         included_types[0] = DS_OBJECT_NAME;
         om_status = om_get(spub_entry->value.object.object,
                OM_EXCLUDE_ALL_BUT_THESE_TYPES+OM_EXCLUDE_SUBOBJECTS,
                included_types, 0, 0, OM_ALL_VALUES,
                &spub_RDNS, &desc_count);

dsX_trace_object(spub_RDNS);
/* Now loop around each RDN, extract a pointer to the AVAS
and then extract the attribute type and value
   */
...
```

If you set the Local Strings parameter on the OM-GET call, the results are converted into the local string syntax (ISO Latin-1). See *OSI-Abstract-Data Manipulation* for more information.

# 2.6. Modifying the Directory Information Base

The interface provides four functions that allow you to modify the Directory:

- Add-Entry
- Modify-Entry
- Remove-Entry
- Modify-RDN

The access control that apply to an entry may prevent modification of that entry, or restrict the modifications that are permitted. Refer to Section 1.7 for information about access control and security.

# 2.6.1. Adding an Entry

Call the Add-Entry function to add an entry to the Directory.

## 2.6.1.1. Specifying the Entry Name

Specify a name for the entry you want to add, as an object of OM class Name. You pass the Name object as the Name parameter of the Add-Entry function.

## 2.6.1.2. Specifying Entry Attributes

You need to supply the Add-Entry function with the details of all the attributes of the entry that you want to add. This is specified as an object of OM class Attribute-List. This object contains all the attribute types and values that you want the entry to contain. You pass the Attribute-List object as the Entry parameter of the Add-Entry function.

# 2.6.2. Modifying an Entry

Call the Modify-Entry function to modify an entry in the Directory. It can be used to add or remove an attribute with all its values, or just a specific attribute value.

You should not modify an entry by deleting it and then adding the new entry. In the time between the deletion and addition of the entry another user could add an entry with the same name.

The Modify-Entry function cannot modify the distinguished value of an entry. Instead, use the Modify-RDN function (see Section 2.6.4).

Note that the access control in operations can restrict a user's ability to alter an entry or attribute. Your application must be able to handle the error returned in these circumstances.

## 2.6.2.1. Specifying the Entry to Be Modified

Specify the name of the entry you want to modify. The name is specified as an object of OM class Name. You pass the Name object as the Name parameter of the Modify-Entry function.

## 2.6.2.2. Specifying the Modifications

You need to supply the details of the modifications you want to make, including the order in which they are to be made. Specify this as a list of attributes and the changes to them. Each modification is an object of the OM class Entry-Mod. See Table 2.1 for details of Entry-Mod.

The list of modifications is an object of the OM class Entry-Mod-List. You pass this object as the Changes parameter of the Modify-Entry function.

**Table 2.1. Attributes of Entry-Mod**

| Attribute | Purpose of the Attribute |
|---|---|
| Attribute-Type | Specifies the attribute type that is the subject of the modification. |
| Attribute-Values | Specifies attribute values of the specified attribute type that are to be added or deleted. |
| Mod-Type | Specifies the modification to be performed on the specified attribute type. See Table 2.2 for details of the possible values of this attribute. |

**Table 2.2. Values of Mod-Type**

| Attribute Value | Modification |
|---|---|
| add-attribute | Adds a new attribute of the type specified in Attribute-Type. |
| add-values | Adds the value or values specified in Attribute-Values to the attribute of the type specified in Attribute-Type. |
| remove-attribute | Removes the attribute of the type specified in Attribute-Type. |
| remove-values | Removes the value or values specified in Attribute-Values from the attribute of the type specified in Attribute-Type. |

You can make any number and combination of modifications with a single call to the Modify-Entry function. The DSA applies the changes in the order they are specified. When the modifications are completed, the entry must conform to the Directory schema. An attempt to perform a modification that does not conform to the schema, for example, adding an invalid attribute value, causes the function to fail and return an error. If the function fails, none of the modifications is made to the entry. For information on the schema, refer to the *DEC X.500 Directory Service Management*.

## 2.6.3. Removing an Entry

Call the Remove-Entry function to remove an entry from the Directory. The entry that you remove must not have any subordinate entries. If you do not know that the entry has no subordinates then use the List function to verify this before calling the Remove-Entry function.

Specify the name of the entry you want to remove from the Directory in the Name parameter of the Remove-Entry function.

The Remove-Entry function has the parameters common to all modification functions, and no others.

## 2.6.4. Modifying a Relative Distinguished Name

Call the Modify-RDN function to change the Relative Distinguished Name (RDN) of an entry.

Specify the name of the entry that you want to change in the Name parameter of the Modify-RDN function.

Specify the new RDN as an object of OM class Relative-Name. You pass this object as the New-RDN parameter of the Modify-RDN function.

You have the option to delete the attribute values that were part of the old RDN but are not part of the new RDN. Use the Delete-Old-RDN parameter of the Modify-RDN function to indicate whether or not such attributes should be deleted:

- If you want to delete all attribute values that were in the old RDN but are not in the new RDN, set Delete-Old-RDN to *true*. If there is one remaining attribute value in an attribute and that is deleted, then the attribute itself is deleted. However, if the attribute is mandatory and the operation would delete it, then the operation fails.

- If you want the old attribute values to remain in the entry, but not as part of the RDN, set Delete-Old-RDN to *false* . If you specify that old values remain and try to add a second value to an attribute that can only have one value, then the operation fails.

The Delete-Old-RDN parameter must have the value *true* if an attribute in the RDN is only permitted to have one value, and has that value changed by the Modify-RDN function.

The return value of this function indicates whether or not the RDN was successfully changed.

# 2.7. Error Handling

A function indicates its successful completion by returning the constant DS_SUCCESS. If an error occurs during the function call then it returns either the constant DS_NO_WORKSPACE or a pointer to a private object that describes the error.

The constant DS_NO_WORKSPACE means that you have not established a valid workspace. Either you have not called the Initialize function, or the call did not succeed and a workspace was not established. Therefore, you need to call the Initialize function before attempting any further function calls.

To interrogate a private object that describes an error, you need to use the OM-Get function to create a public copy of the object. You can then read the value of the attributes of the public object using programming language constructs.

The *OSI-Abstract-Data Manipulation* describes the object classes used to describe errors, and the attributes of those classes.

# Chapter 3. Completing Your Application

This chapter provides details of how to compile and link an application, and other information about completing your application.

## 3.1. Privileges Required on an OpenVMS System

Digital's X.500 API running on an OpenVMS system requires SYSLCK and PRMMBX privileges to communicate with a DSA. If you use Digital's X.500 API to build a directory application, then the application or the user who invokes the application requires those privileges.

## 3.2. Compiling an Application

The XDS component of the DEC X.500 Directory Service contains the following header files:

- xom.h and xomi.h
- xds.h
- xdssap.h
- xdsbdcp.h
- xdsdec.h
- xdsmdup.h and xmhp.h

When you compile an application, you must include the header files xom.h (which automatically includes xomi.h), and xds.h. The header files xdssap.h, xdsbdcp.h, xdsmdup.h (which automatically includes xmhp.h), and xdsdec.h are optional. See Section 3.2.1 for details of what these header files contain.

The location of the header files on an ULTRIX or DEC OSF/1 system is /usr/include. The location of the header files on an OpenVMS system is SYS$LIBRARY.

### 3.2.1. Header Files

The header files xom.h (which automatically includes xomi.h), and xds.h are mandatory. The header files xdssap.h, xdsbdcp.h, xdsmdup.h (which automatically includes xmhp.h), and xdsdec.h are optional. The following sections explain when you need to include them. The order in which you include the optional header files is not important but they must be included after the mandatory header files.

---

**Note**

This release of Digital's X.500 API does not support the object classes defined in the xdsbdcp, xdssap, and xdsmdup packages. That is, the OM classes defined by in these packages are returned in the ASN.1 encoding and there is no OM support for decoding or manipulating them.

The header files xdsbdcp.h, xdssap.h, and xdsmdup.h provide the object identifiers for the directory attributes in these packages. These can be used even though the structured attributes are not supported. Directory attributes which are not objects are supported in the normal way.

### 3.2.1.1. xom.h and xomi.h

The xom.h header file is mandatory, and provides all the necessary declarations for the OSI Object Management (OM) API. This header file is fully described in the *OSI-Abstract-Data Manipulation* .

You must include this header file first, before any other XDS or XOM header files are included.

To include this file, place the following line of code in your application program:

```
#include <xom.h>
```

The xomi.h header file is used by the xom.h header file and therefore does not need to be included separately.

### 3.2.1.2. xds.h

This header file is mandatory, and contains all the necessary declarations for access to directory functions. It declares the interface functions, the structures passed to and from the functions, and the defined constants that are used by the functions and structures.

You must include this header file after the xom.h header file but before any of the optional header files.

To include this file, place the following line of code in your application program:

```
#include <xds.h>
```

### 3.2.1.3. xdsbdcp.h

This optional header file contains the object classes and attribute types supported by the Basic Directory Contents Package. It defines the object identifiers of these directory attribute types and object classes. It also defines the Object Management classes used to represent the values of the attribute types.

Include this header file if your application handles information about people or OSI entities necessary for messaging. The OM classes and attribute types that this package defines are used to hold the following information:

- Names of people, their organizational units and organizations, and the roles they have within those organizations.
- Addresses, including international ISDN, presentation and X.121 addresses.
- Telephone, facsimile, telex and teletex numbers.
- Delivery method to use.
- Details of the DSA, application, or device to contact.

To include this file, place the following line of code in your application program:

```
#include <xdsbdcp.h>
```

### 3.2.1.4. xdsdec.h

This optional header file provides the object classes and attribute types necessary for Digital's extensions to the standard XDS interface. It contains the DSX_PASSWORD attribute type which is used to place a password attribute into a session object. It also contains other attribute types and error codes for communications errors.

To include this file, place the following line of code in your application program:

```
#include <xdsdec.h>
```

### 3.2.1.5. xdssap.h

This optional header file provides the object classes and attribute types supported by the Strong Authentication Package. It defines the object identifiers of these directory attribute types and object classes. It also defines OM classes used to represent the values of the attribute types.

Digital's X.500 Directory Service software does not support strong authentica- tion, so this package is not required.

Include this header file if your application needs to use the authentication of Directory users provided by certification authorities. This is used for the following reasons:

- To check the authentication of other Directory users before allowing them access to information
- To allocate and check authorization for users of your application to perform certain tasks or gain access to certain information
- To provide your application with, or ensure that it has, the necessary authentication to access protected parts of the Directory

To include this file, place the following line of code in your application program:

```
#include <xdssap.h>
```

### 3.2.1.6. xdsmdup.h and xmhp.h

The xdsmdup.h and xmhp.h files are optional and contain the object classes and attribute types supported by the Message Handling System (MHS) Directory User Package. They define the object identifiers of these directory attribute types and object classes. They also also define the Object Management classes used to represent the values of the attribute types.

Include the xdsmdup.h header file if your application needs to access, store, or add to the Directory information concerning an MHS. This includes the following types of information:

- The O/R addresses of MHS users and distribution lists
- The O/R addresses of Message Transfer Agents (MTAs)
- The O/R addresses of Message User Agents
- Characteristics and privileges connected with the O/R addresses

To include this file, place the following line of code in your application program:

```
#include <xdsmdup.h>
```

The xmhp.h header file is used by the xdsmdup.h header file and therefore does not need to be included separately.

### 3.2.2. Compiler

Use an ANSI C compiler to compile your application.

Note that although Digital does not support any other type of C compiler, the "#if_STDC_#else" statements in the header files make it possible to use a non- ANSI C compiler. If you use a non-ANSI C compiler, you will not be able to build public objects statically in your application by declaring a descriptor list. Instead you will have to use the Digital OMX_ macros to create public objects dynamically. These macros are described in *OSI-Abstract-Data Manipulation* .

## 3.3. Linking an Application

### 3.3.1. On a DEC OSF/1 System

On DEC OSF/1 systems, the XDS libraries are installed as both shareable libraries and archive libraries. Existing XDS applications can continue to function without relinking.

By default, when you link an application on a DEC OSF/1 system it links against the shareable libraries. For example, you can link an application as follows:

```
# cc file.c -lxds
```

Note that on DEC OSF/1 systems, this version of the Directory Service also enables you to use the API to develop applications that can use the Cell Directory Service (CDS) of the Distributed Computing Environment (DCE), assuming those directories are installed.

Refer to *Digital DCE for DEC OSF / 1 AXP Product Guide* for details of DCE and CDS, and refer to the XDS and XOM manpages for details of how the API functions support CDS.

If you link an application against shareable libraries, then the link command automatically links to whichever libraries are present. To link using archive libraries, use the following command:

```
# cc -non_shared file.c -lxds -ldxdcds_stub -ldxd \
 -losak -lxti -lxtiosi -lc
```

If you want to use CDS, you cannot link using archive libraries.

Note that an existing X.500 application is unlikely to work correctly with CDS without modification. CDS does not support all of the XDS functions. The XDS and XOM manpages on DEC OSF/1 systems include notes about how to use each function with CDS. Refer to *Digital DCE for DEC OSF / 1 AXP Product Guide* for further details.

Note that the functionality of libraries has not changed on OpenVMS systems or ULTRIX systems, and the option to develop applications for CDS is only available on DEC OSF/1 systems.

#### 3.3.1.1. On an ULTRIX System

You must link your application against the libxds.a directory service library, and the support libraries, libosak.a and libxti.a. The libxds.a library routine provides support for both the directory service routines and object management.

Link to these library routines in the normal way, for example:

```
cc -g xds_read_example.c -lxds -o xds_read_example
```

## 3.3.2. On an OpenVMS System

Link your application in the normal way, specifying the XDS shareable image using an options file, for example:

```
$ link program_name, sys$input:/option
sys$library:dxd$xds_shr/share
sys$library:vaxcrtl.exe/share
^z
$
```

# Appendix A. Digital's X.500 API

Digital's X.500 API is based on Version 1 of the X/Open CAE Specification

*API to Directory Services (XDS)* . This appendix describes Digital extensions to the specification and how optional features in the specification are handled.

For information about extensions and optional features of XDS in a CDS environment, see *Digital DCE for DEC OSF / 1 AXP Product Guide* .

## A.1. Extensions

Digital's X.500 API includes functionality not specified in Version 1 of the X/Open CAE Specification *API to Directory Services (XDS)* . These extensions to the X.500 API are as follows:

- Additional information in the Communications-Error object class
- The XDS public trace routine
- Conversion of local strings to and from T.61 and ISO Latin-1
- A password attribute in the Session object
- Additional error information returned by the Bind function

### A.1.1. The Communications-Error Object Class

The XDS standard defines a single problem code, Communications Error, for all communications errors. Digital's implementation includes additional attributes and problem codes, to provide more information. This information is useful when you are developing an application or interworking with a non-Digital DSA.

### A.1.2. The XDS Public Trace Routine

The xds.h library contains an additional function that allows you to expand the contents of an XOM object. The routine is called dsX_trace_object. The routine takes an OM object as a parameter and prints out, to the current output device, an explanation of what the object contains.

The dsX_trace_object routine has the following features:

- It will fully expand a public object.
- It will tell you the type of a private object.
- It will display details of an error object.
- If a name object or an AVA is encoded in ASN.1, it will display the ASN.1 encoding in both ASCII and hexadecimal.
- It will check for NULL pointers.

To use dsX_trace_object, use the following routine:

```
void dsX_trace_object(OM_object object)
```

See *DEC X.500 Directory Service Programming Reference* for details of this routine.

### A.1.3. Conversion of Local Strings

The X/Open Specification *OSI-Abstract-Data Manipulation API (XOM)* states that strings may be converted to a locally-defined character set. Digital's X.500 API uses the ISO-Latin-1 character set.

### A.1.4. Password in Session Object

The Session OM object has an additional attribute, DSX_PASSWORD, which can be used to control access to the directory.

### A.1.5. Bind Function Error Information

In normal operation, if your application is capable of connecting to CDS as well as an X>500 directory, no error message is returned if the Bind function fails to connect to an X.500 directory, but an error will be returned when your application attempts an X.500 operation. If you require error messages to be returned when the Bind function fails, call the Version function and negotiate the Digital extension feature DSX-RET-X500-BIND-ERR-FTR. See *DEC X.500 Directory Service Programming Reference* for further information.

## A.2. Optional Features

All optional features in the Version 1 of the X/Open CAE Specification *API to Directory Services (XDS)* are supported in Digital's X.500 API, with the following exceptions:

- The ability to encode and decode private objects. This affect s the OM functions OM-Encode and OM-Decode.
- Operations on long strings. This means all directory objects must be able to be stored in memory. This affect s the OM functions for manipulating long strings:
  - OM-Read
  - OM-Write
  - OM-Copy-Value
- Support for Directory Class Definitions contained in the xdsbdcp, xdssap, and xdsmdup packages (see Section 3.2.1).
- See *OSI-Abstract-Data Manipulation* for details of the features of Digital's implementation of the OM API.