

z/OS



DCE

Application Development Guide: Directory Services

z/OS



DCE

Application Development Guide: Directory Services

Note

Before using this information and the product it supports, be sure to read the general information under Appendix A, "Notices" on page 351.

First Edition (March 2001)

This edition, SC24-5906-00, applies to Version 1 Release 1 of z/OS DCE Base Services, z/OS DCE user Data Privacy (DES and CDMF), z/OS DCE User Data Privacy (CDMF) (program number 5694-A01), and to all subsequent releases and modifications until otherwise indicated in new editions.

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

IBM welcomes your comments. A form for reader's comments may be provided at the back of this publication, or you may address your comments to the following address:

International Business Machines Corporation
Information Development, Dept. G60
1701 North Street
Endicott, NY 13760-5553
United States of America

FAX (United States & Canada): 1+607+752-2327
FAX (Other Countries):
Your International Access Code +1+607+752-2327

IBMLink™ (United States customers only): GDLVME(PUBRCF)
Internet e-mail: pubrcf@vnet.ibm.com
World Wide Web: <http://www.ibm.com/servers/eserver/zseries/zos/>

If you would like a reply, be sure to include your name, address, telephone number, or FAX number.

Make sure to include the following in your comment or note:

- Title and order number of this book
- Page number or topic related to your comment

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

© **Copyright International Business Machines Corporation 1994, 2001. All rights reserved.**

Note to U.S. Government Users — Documentation related to restricted rights — Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.

The following statements are provided by the Open Software Foundation.

The information contained within this document is subject to change without notice.

OSF MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL, INCLUDING BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

OSF shall not be liable for errors contained herein, or for any direct or indirect, incidental, special or consequential damages in connection with the furnishing, performance, or use of this material.

Copyright © 1993, 1994 Open Software Foundation, Inc.

This documentation and the software to which it relates are derived in part from materials supplied by the following:

- © Copyright 1990, 1991 Digital Equipment Corporation
- © Copyright 1990, 1991 Hewlett-Packard Company
- © Copyright 1989, 1990, 1991 Transarc Corporation
- © Copyright 1990, 1991 Siemens Nixdorf Informationssysteme AG
- © Copyright 1990, 1991 International Business Machines Corporation
- © Copyright 1988, 1989 Massachusetts Institute of Technology
- © Copyright 1988, 1989 The Regents of the University of California

All Rights Reserved.

Printed in the U.S.A.

THIS DOCUMENT AND THE SOFTWARE DESCRIBED HEREIN ARE FURNISHED UNDER A LICENSE, AND MAY BE USED AND COPIED ONLY IN ACCORDANCE WITH THE TERMS OF SUCH LICENSE AND WITH THE INCLUSION OF THE ABOVE COPYRIGHT NOTICE. TITLE TO AND OWNERSHIP OF THE DOCUMENT AND SOFTWARE REMAIN WITH OSF OR ITS LICENSORS.

Open Software Foundation, OSF, the OSF logo, OSF/1, OSF/Motif, and Motif are trademarks of the Open Software Foundation, Inc.

UNIX is a registered trademark of The Open Group in the United States and other countries.

DEC, DIGITAL, and ULTRIX are registered trademarks of Digital Equipment Corporation.

DECstation 3100 and DECnet are trademarks of Digital Equipment Corporation.

HP, Hewlett-Packard, and LaserJet are trademarks of Hewlett-Packard Company.

Network Computing System and PasswdEtc are registered trademarks of Hewlett-Packard Company.

AFS and Transarc are registered trademarks of the Transarc Corporation.

Episode is a trademark of the Transarc Corporation.

Ethernet is a registered trademark of Xerox Corporation.

DIR-X is a trademark of Siemens Nixdorf Informationssysteme AG.

MX300i is a trademark of Siemens Nixdorf Informationssysteme AG.

NFS, Network File System, SunOS and Sun Microsystems are trademarks of Sun Microsystems, Inc.

X/Open is a trademark of The Open Group in the U.K. and other countries.

PostScript is a trademark of Adobe Systems Incorporated.

FOR U.S. GOVERNMENT CUSTOMERS REGARDING THIS DOCUMENTATION AND THE ASSOCIATED SOFTWARE

These notices shall be marked on any reproduction of this data, in whole or in part.

NOTICE: Notwithstanding any other lease or license that may pertain to, or accompany the delivery of, this computer software, the rights of the Government regarding its use, reproduction and disclosure are as set forth in Section 52.227-19 of the FARs Computer Software-Restricted Rights clause.

RESTRICTED RIGHTS NOTICE: Use, duplication, or disclosure by the Government is subject to the restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 52.227-7013.

RESTRICTED RIGHTS LEGEND: Use, duplication or disclosure by the Government is subject to restrictions as set forth in paragraph (b)(3)(B) of the rights in Technical Data and Computer Software clause in DAR 7-104.9(a). This computer software is submitted with "restricted rights." Use, duplication or disclosure is subject to the restrictions as set forth in NASA FAR SUP 18-52.227-79 (April 1985) "Commercial Computer Software-Restricted Rights (April 1985)." If the contract contains the Clause at 18-52.227-74 "Rights in Data General" then the "Alternate III" clause applies.

US Government Users Restricted Rights—Use, duplication or disclosure restricted by GSA ADP Schedule Contract.

Unpublished—All rights reserved under the Copyright Laws of the United States.

This notice shall be marked on any reproduction of this data, in whole or in part.

Contents

About This Book	xvii
Who Should Use This Book	xvii
DCE Application Development Environment	xvii
Unsupported OSF DCE Functions	xviii
How This Book Is Organized	xix
Terminology Used in This Book	xix
Conventions Used in This Book	xxi
Where to Find More Information	xxi
Softcopy Publications	xxi
Internet Sources	xxii
Using LookAt to Look up Message Explanations	xxii
Accessing Licensed Books on the Web	xxii

Part 1. Using the DCE Directory APIs

1

Chapter 1. DCE Directory Service Overview	3
Using This Book	3
Directory Service Tools	3
Using the DCE Directory Service	3
DCE Directory Service Concepts	4
Structure of DCE Names	6
DCE Name Prefixes	7
Names of Cells	7
CDS Names	8
GDS Names	9
Junctions in DCE Names	9
Application Names	9
The Federated DCE Namespace	9
The GDS Namespace	10
The CDS Namespace	11
Other Namespaces	12
Access to Objects in the Federated DCE Namespace	12
Programming Interfaces to the DCE Directory Service	12
The XDS Interface	13
The RPC Name Service Interface	13
Namespace Junction Interfaces	13

Part 2. CDS Application Programming

15

Chapter 2. Programming in the CDS Namespace	17
Initial Cell Namespace Organization	17
The Cell Profile	18
The LAN Profile	19
The CDS Clearinghouse	19
The Hosts Directory	19
The Subsystems Directory	19
The /: DFS Alias	20
DFS and Security Service Junctions	20
Recommended Use of the CDS Namespace	20

Storing Data in CDS Entries	20
Access Control for CDS Entries	23
Valid Characters and Naming Rules for CDS	25
Metacharacters	27
Additional Rules	27
Maximum Name Sizes	29
Use of Object Identifiers	31
Chapter 3. XDS and the DCE Cell Namespace	33
Introduction to Accessing CDS with XDS	33
Using the Reference Material in this Chapter	33
What You Cannot Do with XDS	34
What Must Be Set Up	34
XDS Objects	34
Object Attributes	36
Interface Objects and Directory Objects	36
Directory Objects and Namespace Entries	38
Values That an Object Can Contain	39
Building a Name Object	39
A Complete Object	41
Class Hierarchy	42
Class Hierarchy and Object Structure	42
Public and Private Objects and XOM	42
XOM Objects and XDS Library Functions	43
Accessing CDS Using the XDS Step-by-Step Procedure	43
Reading and Writing Existing CDS Entry Attributes Using XDS	43
Creating New CDS Entry Attributes	55
Object-Handling Techniques	57
Using XOM to Access CDS	58
Dynamic Creation of Objects	59
XDS/CDS Object Recipes	60
Input XDS/CDS Object Recipes	60
Input Object Classes for XDS/CDS Operations	61
Attribute and Data Type Translation	72
<hr/>	
Part 3. GDS Application Programming	75
Chapter 4. GDS API: Concepts and Overview	77
Directory Service Interfaces	77
The X.500 Directory Information Model	78
Directory Objects	78
Attribute Types	79
Object Identifiers	80
Object Entries	81
X.500 Naming Concepts	83
Distinguished Names	83
Relative Distinguished Names and Attribute Value Assertions	84
Multiple AVAs	84
Aliases	85
Name Verification	86
Schemas	86
The GDS Standard Schema	87
The Structure Rule Table	87

The Object Class Table	89
The Attribute Table	92
Defining Subclasses	93
Abstract Syntax Notation 1	93
ASN.1 Types	94
Basic Encoding Rules	95
Chapter 5. XOM Programming	97
OM Objects	97
OM Object Attributes	97
Object Identifiers	100
C Naming Conventions	100
Public Objects	102
Private Objects	111
Object Classes	111
Packages	117
The Directory Service Package	117
The Basic Directory Contents Package	118
The Strong Authentication Package	119
The Global Directory Service Package	119
The MHS Directory User Package	119
Package Closure	120
Workspaces	120
Storage Management	121
OM Syntaxes for Attribute Values	122
Enumerated Types	123
Object Types	123
Strings	124
Other Syntaxes	124
Service Interface Data Types	124
The OM_descriptor Data Type	125
Data Types for XDS API Function Calls	127
Data Types for XOM API Calls	127
OM Function Calls	128
Summary of OM Function Calls	128
Using the OM Function Calls	129
XOM API Header Files	133
XOM Type Definitions and Symbolic Constant Definitions	133
XOM API Macros	133
Chapter 6. XDS Programming	137
XDS Interface Management Functions	137
The ds_initialize() Function Call	138
The ds_version() Function Call	138
The ds_shutdown() Function Call	140
Directory Connection Management Functions	140
A Directory Session	140
The ds_bind() Function Call	140
The ds_unbind() Function Call	141
Automatic Connection Management	141
XDS Interface Class Definitions	141
The DS_C_CONTEXT Parameter	142
Directory Class Definitions	142
Directory Operation Functions	143

Directory Read Operations	143
Reading an Entry from the Directory	144
Step 1: Export Object Identifiers for Required Directory Classes and Attributes	144
Step 2: Declare Local Variables	145
Step 3: Build Public Objects	145
Step 4: Create an Entry-Information-Selection Parameter	146
Step 5: Perform the Read Operation	147
Directory Search Operations	150
Directory Modify Operations	150
Modifying Directory Entries	151
Step 1: Export Object Identifiers for Required Directory Classes and Attributes	152
Step 2: Declare Local Variables	152
Step 3: Build Public Objects	153
Step 4: Create Descriptor Lists for Attributes	154
Step 5: Perform the Operations	155
Return Codes	157
Chapter 7. Example Application Programs	159
General Programming Guidelines	159
The example.c Program	159
The example.c Code	162
Error Handling	168
The teldir.c Program	170
Predefined Static Public Objects	170
Partially Defined Static Public Objects	171
Dynamically Defined Public Objects	172
Main Program Procedural Steps	173
The teldir.c Code	174
Chapter 8. Using Threads with the XDS/XOM API	191
Overview of Example Threads Program	192
User Interface	193
Input File Format	193
Program Output	193
Prerequisites	194
Description of Thradd Example Program	194
Detailed Description of Thread Specifics	195
The thradd.c Code	197
The thradd.h Header File	205
Chapter 9. XDS/XOM Convenience Routines	209
String Handling	209
Strings Representing GDS Attribute Information	210
Strings Representing Structured GDS Attribute Information	210
Strings Representing a Structured GDS Attribute Value	212
Strings Representing a Distinguished Name	212
Strings Representing Expressions	213
Examples of strings handled by omX_string_to_object()	214
Examples of strings returned by omX_object_to_string()	216
The teldir2.c Program	217
The teldir2.c Code	218
<hr/>	
Part 4. XDS/XOM Supplementary Information	231

Chapter 10. XDS Interface Description	233
XDS Conformance to Standards	233
The XDS Functions	234
The XDS Negotiation Sequence	235
The session Parameter	235
The context Parameter	236
The XDS Function Arguments	236
Attribute and Attribute Value Assertion	237
The Entry-Information-Selection Parameter	237
The name Parameter	238
XDS Function Call Results	238
The invoke-id Parameter	238
The result Parameter	238
The DS_status Return Value	239
Synchronous Operations	239
Security and XDS	240
Other Features of the XDS Interface	240
Automatic Connection Management	240
Automatic Continuation and Referral Handling	240
Chapter 11. XDS Class Definitions	241
Introduction to OM Classes	241
XDS Errors	241
OM Class Hierarchy	242
DS_C_ABANDON_FAILED	244
DS_C_ACCESS_POINT	245
DS_C_ADDRESS	245
DS_C_ATTRIBUTE	245
DS_C_ATTRIBUTE_ERROR	246
DS_C_ATTRIBUTE_LIST	246
DS_C_ATTRIBUTE_PROBLEM	247
DS_C_AVA	248
DS_C_COMMON_RESULTS	248
DS_C_COMMUNICATIONS_ERROR	248
DS_C_COMPARE_RESULT	249
DS_C_CONTEXT	249
DS_C_CONTINUATION_REF	252
DS_C_DS_DN	253
DS_C_DS_RDN	253
DS_C_ENTRY_INFO	254
DS_C_ENTRY_INFO_SELECTION	254
DS_C_ENTRY_MOD	255
DS_C_ENTRY_MOD_LIST	256
DS_C_ERROR	256
DS_C_EXT	258
DS_C_FILTER	259
DS_C_FILTER_ITEM	260
DS_C_LIBRARY_ERROR	261
DS_C_LIST_INFO	262
DS_C_LIST_INFO_ITEM	263
DS_C_LIST_RESULT	264
DS_C_NAME	264
DS_C_NAME_ERROR	265
DS_C_OPERATION_PROGRESS	265

DS_C_PARTIAL_OUTCOME_QUAL	266
DS_C_PRESENTATION_ADDRESS	267
DS_C_READ_RESULT	268
DS_C_REFERRAL	268
DS_C_RELATIVE_NAME	268
DS_C_SEARCH_INFO	268
DS_C_SEARCH_RESULT	269
DS_C_SECURITY_ERROR	270
DS_C_SERVICE_ERROR	270
DS_C_SESSION	271
DS_C_SYSTEM_ERROR	272
DS_C_UPDATE_ERROR	273
Chapter 12. Basic Directory Contents Package	275
Selected Attribute Types	275
Selected Object Classes	282
OM Class Hierarchy	283
DS_C_FACSIMILE_TELEPHONE_NUMBER	284
DS_C_POSTAL_ADDRESS	284
DS_C_SEARCH_CRITERION	285
DS_C_SEARCH_GUIDE	286
DS_C_TELETEX_TERMINAL_IDENTIFIER	286
DS_C_TELEX_NUMBER	287
Chapter 13. Strong Authentication Package	289
SAP Attribute Types	289
Strong Authentication Package Object Classes	291
OM Class Hierarchy	291
DS_C_ALGORITHM_IDENT	291
DS_C_CERT	292
DS_C_CERT_LIST	293
DS_C_CERT_PAIR	293
DS_C_CERT_SUBLIST	294
DS_C_SIGNATURE	294
Chapter 14. MHS Directory User Package	297
MDUP Attribute Types	297
MDUP Object Classes	299
MDUP OM Class Hierarchy	300
MH_C_OR_ADDRESS	300
MH_C_OR_NAME	310
DS_C_DL_SUBMIT_PERMS	310
Chapter 15. Global Directory Service Package	313
GDSP Attribute Types	313
GDSP Object Classes	316
GDSP OM Class Hierarchy	316
DSX_C_GDS_ACL	316
DSX_C_GDS_ACL_ITEM	317
DSX_C_GDS_CONTEXT	318
DSX_C_GDS_SESSION	321
Chapter 16. Distributed Management Environment Support	323
DME Attribute Types	323

DME Object Classes	324
Chapter 17. Information Syntaxes	325
Syntax Templates	325
Syntaxes	325
Strings	326
Representation of String Values	327
Relationship to ASN.1 Simple Types	327
Relationship to ASN.1 Useful Types	327
Relationship to ASN.1 Character String Types	328
Relationship to ASN.1 Type Constructors	328
Chapter 18. XOM Service Interface	331
Standards Conformance	331
XOM Data Types	331
OM_boolean	333
OM_descriptor	333
OM_enumeration	334
OM_exclusions	334
OM_integer	335
OM_modification	335
OM_object	335
OM_object_identifier	335
OM_private_object	337
OM_public_object	337
OM_return_code	337
OM_string	338
OM_syntax	339
OM_type	339
OM_type_list	340
OM_value	340
OM_value_length	341
OM_value_position	341
OM_workspace	341
XOM Functions	341
XOM Return Codes	343
Chapter 19. Object Management Package	347
Class Hierarchy	347
Class Definitions	347
OM_C_ENCODING	347
OM_C_EXTERNAL	348
OM_C_OBJECT	349
Appendix A. Notices	351
Trademarks	352
Programming Interface Information	353
Glossary	355
Bibliography	373
z/OS DCE Publications	373
z/OS SecureWay® Security Server Publications	373
Tool Control Language Publication	374

IBM C/C++ Language Publication	374
z/OS DCE Application Support Publications	374
Encina Publications	375
Index	377

Figures

1.	A Federated DCE Namespace	10
2.	GDS Namespace Entries and Directory Objects	11
3.	The Cell Namespace after Configuration	18
4.	A Possible Namespace Structure	22
5.	Valid Characters in CDS, GDS, and DNS Names	26
6.	T61 Syntax	30
7.	Combinations of Diacritical Characters and Basic Letters	31
8.	One Object Descriptor	35
9.	A Complete Object Represented	35
10.	A Three-Layer Compound Object	36
11.	Directory Objects and XDS Interface Objects	37
12.	Directory Objects and Namespace Entries	38
13.	DS_C_READ_RESULT Object Structure	50
14.	DS_C_ENTRY_INFO Object Structure	52
15.	DS_C_ATTRIBUTE Object Structure	54
16.	DS_C_ATTRIBUTE_LIST Object	63
17.	DS_C_DS_DN Object	65
18.	DS_C_ENTRY_MOD_LIST Object	68
19.	The DS_C_ENTRY_INFO_SELECTION Object	71
20.	XDS: Interface to GDS and CDS	78
21.	The Structure of the DIB	79
22.	Object Identifiers	80
23.	A Directory Entry Describing Organizational Person	82
24.	A Distinguished Name in a Directory Information Tree	83
25.	An Alias in the Directory Information Tree	85
26.	A Subtree Populated by Aliases	86
27.	SRT DIT Structure for the GDS Standard Schema	88
28.	A Partial Representation of the Object Class Table	90
29.	The Relationship Between Schemas and the DIT	93
30.	Mapping the Class Definition of DS_C_ENTRY_INFO_SELECTION	99
31.	A Representation of a Public Object Using a Descriptor List	103
32.	A Descriptor List for the Public Object: country	104
33.	The Distinguished Name of Peter Piper in the DIT	105
34.	Building a Distinguished Name	108
35.	A Simplified View of the Structure of a Distinguished Name	109
36.	Client-Generated and Service-Generated Objects	110
37.	The OM Class DS_C_ENTRY_INFO_SELECTION	112
38.	A Comparison of Two Classes With and Without an Abstract OM Class	113
39.	A Complete Description of the Concrete OM Class DS_C_ATTRIBUTE	116
40.	Data Type: OM_descriptor_struct	125
41.	Initializing Descriptors	126
42.	An Object and a Subordinate Object	127
43.	Extracting Information Using om_get()	132
44.	Output from ds_read(): DS_C_READ_RESULT	149
45.	Sample Directory Tree	151
46.	OM Class DS_C_LIST_RESULT	156
47.	Issuing XDS/XOM Calls from within Different Threads	191
48.	Program Flow for the thradd Sample Program	195
49.	OM_String Elements	338

Tables

1.	Metacharacters and Their Meaning	27
2.	Summary of CDS, GDS, and DNS Characteristics	27
3.	Maximum Sizes of Directory Service Names	29
4.	Directory Service Functions with their Required Input Objects	61
5.	CDS Attributes to OM Syntax Translation	72
6.	OM Syntax to CDS Data Types Translation	72
7.	CDS Data Types to OM Syntax Translation	73
8.	Object Identifiers for Selected Attribute Types	80
9.	Structure Rule Table Entries	87
10.	Object Class Table Entries	89
11.	Object Identifiers for Selected Classes	90
12.	Attribute Table Entries	92
13.	Syntax for the Simple ASN.1 Types	94
14.	C Naming Conventions for XDS	101
15.	C Naming Conventions for XOM	102
16.	Comparison of Private and Public Objects	111
17.	Description of an OM Attribute Using Syntax Enum(*)	123
18.	Description of an OM Attribute with Syntax Object(*)	123
19.	Mapping of XDS API Functions to the Abstract Services	143
20.	The XDS Interface Functions	234
21.	OM Attributes of DS_C_ACCESS_POINT	245
22.	OM Attributes of DS_C_ATTRIBUTE	245
23.	OM Attributes of DS_C_ATTRIBUTE_ERROR	246
24.	OM Attributes of DS_C_ATTRIBUTE_LIST	246
25.	OM Attributes of DS_C_ATTRIBUTE_PROBLEM	247
26.	OM Attributes of DS_C_COMMON_RESULTS	248
27.	OM Attributes of DS_C_COMPARE_RESULT	249
28.	OM Attributes of DS_C_CONTEXT	249
29.	OM Attributes of DS_C_CONTINUATION_REF	252
30.	OM Attribute of DS_C_DS_DN	253
31.	OM Attribute of DS_C_DS_RDN	254
32.	OM Attributes of DS_C_ENTRY_INFO	254
33.	OM Attributes of DS_C_ENTRY_INFO_SELECTION	255
34.	OM Attributes of DS_C_ENTRY_MOD	255
35.	OM Attributes of DS_C_ENTRY_MOD_LIST	256
36.	OM Attributes of DS_C_ERROR	256
37.	OM Attributes of DS_C_EXT	258
38.	OM Attributes of DS_C_FILTER	259
39.	OM Attributes of DS_C_FILTER_ITEM	260
40.	OM Attributes of DS_C_LIST_INFO	262
41.	OM Attributes of DS_C_LIST_INFO_ITEM	263
42.	OM Attributes of DS_C_LIST_RESULT	264
43.	OM Attributes of DS_C_NAME_ERROR	265
44.	OM Attributes of DS_C_OPERATION_PROGRESS	266
45.	OM Attributes of DS_C_PARTIAL_OUTCOME_QUAL	266
46.	OM Attributes of DS_C_PRESENTATION_ADDRESS	267
47.	OM Attributes of DS_C_READ_RESULT	268
48.	OM Attributes of DS_C_SEARCH_INFO	269
49.	OM Attributes of DS_C_SEARCH_RESULT	269
50.	OM Attributes of DS_C_SESSION	271

51.	Object Identifiers for Selected Attribute Types	276
52.	Representation of Values for Selected Attribute Types	277
53.	Object Identifiers for Selected Object Classes	283
54.	OM Attributes of DS_C_FACSIMILE_PHONE_NBR	284
55.	OM Attributes of DS_C_POSTAL_ADDRESS	284
56.	OM Attributes of DS_C_SEARCH_CRITERION	285
57.	OM Attributes of DS_C_SEARCH_GUIDE	286
58.	OM Attributes of DS_C_TELETEX_TERM_IDENT	286
59.	OM Attributes of DS_C_TELEX_NBR	287
60.	Object Identifiers for SAP Attribute Types	290
61.	Representation of Values for SAP Attribute Types	290
62.	Object Identifiers for SAP Object Classes	291
63.	OM Attributes of DS_C_ALGORITHM_IDENT	291
64.	OM Attributes of DS_C_CERT	292
65.	OM Attributes of DS_C_CERT_LIST	293
66.	OM Attributes of DS_C_CERT_PAIR	293
67.	OM Attributes of DS_C_CERT_SUBLIST	294
68.	OM Attributes of DS_C_SIGNATURE	294
69.	Object Identifiers for MDUP Attribute Types	297
70.	Representation of Values for MDUP Attribute Types	298
71.	Object Identifiers for MDUP Object Classes	299
72.	Attributes Specific to MH_C_OR_ADDRESS	300
73.	Forms of Originator/Recipient Address	307
74.	Attribute Specific to MH_C_OR_NAME	310
75.	OM Attributes of DS_C_DL_SUBMIT_PERMS	310
76.	Object Identifiers for GDSP Attribute Types	313
77.	Representation of Values for GDSP Attribute Types	314
78.	Object Identifiers for GDSP Object Classes	316
79.	OM Attributes of DSX_C_GDS_ACL	316
80.	OM Attributes of DSX_C_GDS_ACL_ITEM	317
81.	OM Attributes of a DSX_C_GDS_CONTEXT	318
82.	Default DSX_C_GDS_CONTEXT	320
83.	OM Attributes of DSX_C_GDS_SESSION	321
84.	Default DSX_C_GDS_SESSION	321
85.	Object Identifier for DME Attribute Type	323
86.	Representation of Values for DME Attribute Types	324
87.	Object Identifier for DME Object Class	324
88.	String Syntax Identifiers	326
89.	Syntax for ASN.1's Simple Types	327
90.	Syntax for ASN.1's Useful Types	327
91.	Syntax for ASN.1's Character String Types	328
92.	Syntaxes for ASN.1's Type Constructors	328
93.	XOM Service Interface Data Types	331
94.	Assigning Meanings to Values	340
95.	XOM Service Interface Functions	341
96.	OM Functions and Their Corresponding Abbreviation	343
97.	XOM Service Interface Return Codes	343
98.	Attributes Specific to OM_C_Encoding	347
99.	Attributes Specific to OM_C_External	348
100.	Attributes Specific to OM_C_Object	349

About This Book

The objective of this book is to assist you in designing, writing, compiling, linking, and running distributed applications on the IBM z/OS operating system. Specifically, use this book for creating applications with DCE running on the stand-alone z/OS system. The steps to develop a distributed application using DCE services and application programming interfaces (API) are described in progressive detail. Also discussed are the development decisions and tools that you need to consider when developing your distributed application using z/OS DCE.

To create DCE applications that access IMS™ or CICS® transactions, refer to *z/OS DCE Application Support Programming Guide*.

Who Should Use This Book

This book assumes you are an experienced application developer or programmer with a working knowledge of the C programming language and the z/OS operating system. You do not have to possess prior knowledge of, or experience with, designing and writing distributed applications using the Open Software Foundation (OSF) Distributed Computing Environment (DCE) services and APIs.

Ideally, you should be able to:

- Allocate z/OS data sets
- Edit, browse, and copy z/OS data sets and associated members
- Print data sets
- Write and submit batch jobs on z/OS
- Write, compile, link, and run C/C++ programs on z/OS
- Write and understand JCL to run on z/OS
- Understand Shell and TSO/E commands.

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

- Interactive System Productivity Facility/Program Development Facility (ISPF/PDF)
- Concepts behind a distributed application
- Using the Spool Display and Search Facility (SDSF) to check on the status of your application.

Some exposure to the UNIX or AIX® operating system is helpful but not essential to use this book.

You should be familiar with the concepts of the Distributed Computing Environment. If you are not, read *z/OS DCE Introduction*.

DCE Application Development Environment

It is conceivable that you may develop your DCE applications on a platform other than the z/OS operating system. Perhaps you may prefer to work on a UNIX-based workstation or a proprietary operating system. If your goal is to ultimately run either the client or server portion of your DCE application on z/OS, ensure that portion of your DCE application conforms to all recommendations contained in this book.

This book describes the development steps assuming you are developing your DCE applications on the z/OS operating system. If you are developing DCE applications on the z/OS platform that are targeted to run on another platform, consult the DCE application development documentation associated with that platform.

Unsupported OSF DCE Functions

The following DCE technology functions, which may be available in the Distributed Computing Environment product from OSF or on DCE offerings from other vendors, are not supported in z/OS DCE:

- DCE Directory Services
 - X/Open Data Services (XDS) function (Global Directory Service (GDS) portion, DME routines)
 - X/Open OSI-Abstract-Data Manipulation (XOM) function (GDS portion)
 - Global Directory

On z/OS, only CDS, XDS, and XOM access to CDS are supported. GDS, XDS, and XOM access to GDS are not supported.

The following DCE daemon is not supported on z/OS DCE:

- DCE Security daemon

OSF DCE Programming Interfaces: The following programming interfaces are not supported:

- pthread interfaces
 - The following interfaces are not supported by z/OS DCE and return -1, errno ENOSYS:
 - pthread_attr_getinheritsched()
 - pthread_attr_getprio()
 - pthread_attr_getsched()
 - pthread_attr_setinheritsched()
 - pthread_attr_setprio()
 - pthread_attr_setsched()
 - pthread_getprio()
 - pthread_getscheduler()
 - pthread_setprio()
 - pthread_setscheduler()
 - For all pthread interfaces (including mutexes, threads, condition variables and so on), the interfaces do not accept copies of the objects as a parameter. The object returned from the pthread interface to create the object must be used at all times.
 - Unlike the OSF DCE implementation, the z/OS DCE implementation of the following functions can raise an exception (**exc_e_cpa_error**) in error situations:
 - pthread_lock_global_np()
 - pthread_unlock_global_np()
 - pthread_cond_timedwait() expects an absolute hardware time (that is, time-of-day clock value) for the wait time instead of the DCE software clock time, which is what OSF/DCE expects. pthread_get_expiration_np() returns a software adjusted time as in the OSF/DCE model, and is used as input to pthread_cond_timedwait().
 - exc_report() does not print out a message to stderr as expected. z/OS DCE uses Reliability, Availability and Serviceability (RAS) services to log messages instead of this function.
- Exceptions
 - z/OS DCE catches z/OS abends in addition to the set of predefined exceptions and user defined exceptions.
 - TRY/CATCH/ENDTRY macros can raise an exc_e_insmem exception if they cannot get enough heap storage.
 - TRY/CATCH/ENDTRY macros can raise an exc_e_uninitexc exception if they detect that the CATCH does not specify a valid exception.

- Remote Procedure Call
 - `rpc_mgmt_set_server_stack_size()`
- Security Services
 - `sec_login_get_pwent()`
 - `sec_login_init_first()`

How This Book Is Organized

This guide describes how application developers can access the DCE Directory Service. From the application programmer's perspective, the Directory Service has the following main parts:

- The DCE Cell Directory Service (CDS)
- GDS
- XDS and XOM programming interfaces.

This is reflected in the organization of the book:

- Part 1, "Using the DCE Directory APIs" on page 1
- Part 2, "CDS Application Programming" on page 15
- Part 3, "GDS Application Programming" on page 75
- Part 4, "XDS/XOM Supplementary Information" on page 231

Part 2, CDS Application Programming and Part 3, GDS Application Programming contain conceptual material on CDS and GDS with descriptions of programming tasks, including the use of programming interfaces. Chapters in each of these parts contain annotated source code for sample applications.

Part 4, XDS/XOM Supplementary Information consists mostly of tables of values for the data structures used by the XDS and XOM application interfaces, which are the interfaces used to directly access the DCE Directory Service.

To find more information on topics related to application development not addressed in this book, consult the following:

- *z/OS DCE Application Development Reference*, SC24-5908
- *z/OS DCE Administration Guide*, SC24-5904
- *z/OS DCE Application Support Programming Guide*, SC24-5902 (CICS and IMS)
- *z/OS DCE Messages and Codes*, SC24-5912

For example, the DCE CDS is discussed in detail as a separate component in the administration documentation. Similarly, certain aspects of the DCE Security Service important to application developers (such as adding new principals to the registry database) are found only in the administration books.

Terminology Used in This Book

Because DCE technology has been developed from the UNIX environment, many DCE concepts and terms contained herein relate to that environment. z/OS terms and concepts are used throughout this book wherever possible.

The following table explains how certain terms are used in this book and how they are related.

Related Terms	Relationship
<p>file</p> <p>data set</p> <p>sequential data set</p> <p>partitioned data set member</p> <p>hierarchical file system (HFS) file</p>	<p>Throughout this book, the term <i>file</i> can refer to a sequential data set, a member of a partitioned data set, or a hierarchical file system (HFS) file. For more information on hierarchical file systems in z/OS, see <i>z/OS UNIX System Services User's Guide</i>, SA22-7801.</p>
<p>user prefix</p> <p>data set names</p>	<p>The term <i>user prefix</i> is used throughout this book when referring to the names of data sets in a TSO/E environment. In that environment, the user prefix is usually a user's logon identification. If desired, you can set the user prefix to a value other than the your logon identification by using the TSO/E PROFILE command. In z/OS batch mode, your user prefix depends on whether Resource Access Control Facility (RACF®), a component of the SecureWay® Security Server for z/OS, or another security product is installed on your system. If RACF is installed, and you are processing in batch mode, your user prefix can be the same as your logon user identification. If RACF is not installed and you are processing in batch mode under z/OS, you may not have to use a prefix. See your systems programmer to determine the RACF settings for your site.</p> <p>Unless otherwise specified, when the full name of a data set is referred to, the high-level qualifier for that data set will be represented by <i>USERPRFX</i>. The <i>USERPRFX</i> is determined by the application developer, and depends on the library where the application is installed. For example, <i>USERPRFX.EXAMPLE.C(MEMBER)</i> represents a partitioned data set whose first-level qualifier is represented by <i>USERPRFX</i>, whose second-level qualifier is <i>EXAMPLE</i>, and whose third-level qualifier is <i>C</i>. Its member is <i>MEMBER</i>.</p>
<p>application programming interface (API)</p> <p>call</p> <p>function</p> <p>routine</p>	<p>Throughout this book, the terms <i>API</i>, <i>call</i>, <i>function</i>, and <i>routine</i> all refer to the same z/OS DCE application programming interface. For example, rpc_binding_free() API, rpc_binding_free() call, and rpc_binding_free() routine, all refer to the same rpc_binding_free() function.</p>
<p>DCE components</p>	<p>Throughout this book, all references to individual DCE components (such as RPC) refer to that component with z/OS DCE. For example, references to RPC, DCE RPC, and z/OS DCE RPC all refer to the same z/OS DCE component.</p>
<p>z/OS SecureWay Security Server DCE</p>	<p>In this book the term "DCE Security Server" (or simply "Security Server") refers to the z/OS SecureWay Security Server DCE or to a DCE Security Server provided on another host in the DCE cell. The z/OS SecureWay Security Server DCE is a component of the SecureWay Security Server for z/OS.</p>
<p>daemon</p> <p>process</p> <p>started task</p> <p>address space</p>	<p>The term <i>daemon</i> (originating from the UNIX operating system) is used throughout this book. It is synonymous with a process. Usually there is one process per address space, however the DCEKERN started task is an exception as its address space contains five processes (or daemons).</p>

Conventions Used in This Book

This book uses the following typographic conventions:

Bold	Bold words or characters represent system elements that you must enter into the system literally, such as commands, options, or path names.
<i>Italic</i>	<i>Italic</i> words or characters represent values for variables.
Example font	Examples and information displayed by the system appear in constant width type style.
[]	Brackets enclose optional items in format and syntax descriptions.
{ }	Braces enclose a list from which you must choose an item in format and syntax descriptions.
	A vertical bar separates items in a list of choices.
< >	Angle brackets enclose the name of a key on the keyboard.
...	Horizontal ellipsis points indicate that you can repeat the preceding item one or more times.
\	A backslash is used as a continuation character when entering commands from the shell that exceed one line (255 characters). If the command exceeds one line, use the backslash character \ as the last non-blank character on the line to be continued, and continue the command on the next line.

This book uses the following keying conventions:

<Alt-c>	The notation <Alt-c> followed by the name of a key indicates a control character sequence.
<Return>	The notation <Return> refers to the key on your keyboard that is labeled with the word Return or Enter, or with a left arrow.
Entering commands	When instructed to enter a command, type the command name and then press <Return>.

Where to Find More Information

Where necessary, this book references information in other books using shortened versions of the book title. For complete titles and order numbers of the books for all products that are part of z/OS, see the *z/OS Information Roadmap*, SA22-7500. For complete titles and order numbers of the books for z/OS DCE, refer to the publications listed in the “Bibliography” on page 373.

For information about installing z/OS DCE components, see the *z/OS Program Directory*.

Softcopy Publications

The z/OS DCE library is available on a CD-ROM, *z/OS Collection*, SK3T-4269. The CD-ROM online library collection is a set of unlicensed books for z/OS and related products that includes the IBM Library Reader.™ This is a program that enables you to view the BookManager® files. This CD-ROM also contains the Portable Document Format (PDF) files. You can view or print these files with the Adobe Acrobat reader.

Internet Sources

The Softcopy z/OS publications are also available for web-browsing and for viewing or printing PDFs using the following URL:

<http://www.ibm.com/servers/eserver/zseries/zos/bkserv/>

You can also provide comments about this book and any other z/OS documentation by visiting that URL. Your feedback is important in helping to provide the most accurate and high-quality information.

Using LookAt to Look up Message Explanations

LookAt is an online facility that allows you to look up explanations for z/OS messages. You can also use LookAt to look up explanations of system abends.

Using LookAt to find information is faster than a conventional search because LookAt goes directly to the explanation.

LookAt can be accessed from the Internet or from a TSO command line.

You can use LookAt on the Internet at:

<http://www.ibm.com/servers/eserver/zseries/zos/bkserv/lookat/lookat.html>

To use LookAt as a TSO command, LookAt must be installed on your host system. You can obtain the LookAt code for TSO from the LookAt Web site by clicking on the **News and Help** link or from the *z/OS Collection*, SK3T-4269.

To find a message explanation from a TSO command line, simply enter: **lookat** *message-id* as in the following:

```
lookat iec192i
```

This results in direct access to the message explanation for message IEC192I.

To find a message explanation from the LookAt Web site, simply enter the message ID and select the release with which you are working.

Note: Some messages have information in more than one book. For example, IEC192I has routing and descriptor codes listed in *z/OS MVS Routing and Descriptor Codes*, SA22-7624. For such messages, LookAt prompts you to choose which book to open.

Accessing Licensed Books on the Web

z/OS licensed documentation in PDF format is available on the Internet at the IBM Resource Link site:

<http://www.ibm.com/servers/resourceLink>

Licensed books are available only to customers with a z/OS license. Access to these books requires an IBM Resource Link user ID, password, and z/OS licensed book key code. The z/OS order that you received provides a memo that includes your key code.

To obtain your IBM Resource Link user ID and password, logon to:

<http://www.ibm.com/servers/resourceLink>

To register for access to the z/OS licensed books:

1. Logon to Resource Link using your Resource Link user ID and password.
2. Select **User Profiles** located on the left-hand navigation bar.
3. Select **Access Profile**.
4. Select **Request Access to Licensed books**.
5. Supply your key code where requested and select the **Submit** button.

If you supplied the correct key code you will receive confirmation that your request is being processed.

After your request is processed you will receive an e-mail confirmation.

Note: You cannot access the z/OS licensed books unless you have registered for access to them and received an e-mail confirmation informing you that your request has been processed.

To access the licensed books:

1. Logon to Resource Link using your Resource Link user ID and password.
2. Select **Library**.
3. Select **zSeries**.
4. Select **Software**.
5. Select **z/OS**.
6. Access the licensed book by selecting the appropriate element.

Part 1. Using the DCE Directory APIs

This part describes how application developers access the DCE Directory Service.

Chapter 1. DCE Directory Service Overview	3	The Federated DCE Namespace	9
Using This Book	3	The GDS Namespace	10
Directory Service Tools	3	The CDS Namespace	11
Using the DCE Directory Service	3	Other Namespaces	12
DCE Directory Service Concepts	4	Access to Objects in the Federated DCE	
Structure of DCE Names	6	Namespace	12
DCE Name Prefixes	7	Programming Interfaces to the DCE Directory	
Names of Cells	7	Service	12
CDS Names	8	The XDS Interface	13
GDS Names	9	The RPC Name Service Interface	13
Junctions in DCE Names	9	Namespace Junction Interfaces	13
Application Names	9		

Chapter 1. DCE Directory Service Overview

This chapter is an overview of the DCE Directory Service for application developers. It introduces DCE Directory Service concepts and describes the structure of DCE names and the DCE namespace. Finally, an overview of the APIs used to access the DCE Directory Services is provided.

Using This Book

Before reading this book, you should have read the *z/OS DCE Introduction*. It contains overviews, with illustrations, of all the DCE components and of DCE as a whole. Many concepts and details are explained that are necessary for a complete understanding of what is described in this book.

At this point, you are ready to begin programming and can proceed to the chapters on XDS/XOM supplementary information beginning at Chapter 10, “XDS Interface Description” on page 233. The main purpose of this information is to serve as a convenient location to look up the details of object values and structures required for coding applications.

If you do not find the information you need in either this guide or in *z/OS DCE Application Development Reference*, see *z/OS DCE Administration Guide* and *z/OS DCE Command Reference*. Information that is of interest to application developers may be found in the DCE Administration documentation.

Directory Service Tools

Both CDS and GDS have commands that allow system administrators to inspect and alter the contents of the directory. These can be useful when developing applications that access the DCE namespace.

For information on performing administrative functions for CDS, see the “Managing the DCE Directory Service” section in the *z/OS DCE Administration Guide*, which describes the DCE and CDS control programs (**dcecp** and **cdscp**).

Note: GDS commands are not supported by z/OS DCE.

Using the DCE Directory Service

The DCE Directory Service can be used in many ways. It is used by the other DCE services to support the DCE environment. For example, cells are registered in the global part of the Directory Service, enabling users from different cells to share information and resources.

The DCE Directory Service is also useful to DCE applications. The client and server sides of an application can use the Directory Service to find each other’s locations. The Directory Service can also be used to store information that needs to be available in a globally accessible, well-known place.

For example, one DCE application could be a print service consisting of a client side application that makes requests for jobs to be printed, and a server side that prints jobs on an available printer. The Directory Service could be used as a central place where the print clients could look up the location of a print server. Furthermore, the Directory Service could be used to store information about printers, for example, the type of jobs a printer can accept, its current status, and its load.

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

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

The Directory Service can also secure data that is kept in a globally accessible place. It supports Access Control Lists (ACLs) that control who can read, modify, create, and perform other operations on its data.

As you learn about the DCE Directory Service and how to access it, think about the ways your applications can take advantage of the services it provides.

DCE Directory Service Concepts

This section describes DCE Directory Service concepts that are important to application developers. Concepts specific to GDS are covered in more detail in Chapter 10, “XDS Interface Description” on page 233. The following concepts are intended to convey general definitions that are applicable to the DCE Directory Service as a whole rather than specific to a particular Directory Service component. For more detailed definitions, see “Glossary” on page 355.

- DCE Namespace

The DCE namespace is the collection of names in a DCE environment. It can be made up of several domains, in which different types of servers own the names in different parts of the namespace. (See “The Federated DCE Namespace” on page 9.) Typically, there are two high-level, or global, domains to a DCE namespace: the GDS namespace and the Domain Name Service (DNS) namespace. At the next level is the CDS namespace, with names contained in the cell’s CDS Server. A DCE environment always contains a cell namespace, which is implemented by CDS. Parts of the DCE namespace may not be contained in any of the Directory Services; for example, the Security namespace contains principals and groups contained in the Security Server.

The term *DCE namespace* is used when referring to names, but not the information associated with them. For example, it would include the name of a printer in the Directory Service, but not its associated location attribute. The DCE namespace refers to all the names in the DCE environment. You can refer to an object and its contents by using its name.

- Cell Namespace

All of the names found in a single DCE cell comprise the cell’s namespace. These include names managed by the cell’s CDS Server and Security Server, and any other names that reside within a particular cell.

- Hierarchy

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

- Directory

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

- Directory Service

A directory service is software that manages names and their associated attributes. A directory service can store information, be queried about information, and be requested to change information. DCE contains two different directory services: CDS and GDS. It also interacts with a third directory service, DNS, which is not part of DCE.

- Junction

A junction is a specialized entry in the DCE namespace that contains binding information to enable communications between different DCE services. For example, the point where the DCE Security entries are mounted into a CDS namespace is a junction.

- Object

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

- Entry

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

- Class

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

- Link

A link is one type of object class. This type of object is a pointer to another object. A CDS link is similar to a GDS alias.

- Attribute

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

- Type

Every attribute is characterized as being of a certain type. The attribute is used to hold a certain kind of data, such as a zip code or the name of a country. An attribute type indicates the class of information given by an attribute. Examples of X.500 attribute types include: common name, country name, organization name, street address, postal code, member, and presentation address. Entries can also be classified by type; for entries, the term used is **class**.

- Value

An attribute can have one or more values. The value is an instance of the class of information indicated by the type. For example, M2B 4Y6 is an instance of a postal code.

- Object Identifier

Directory attributes are uniquely identified by Object Identifiers (OIDs), which are administered by the International Organization for Standardization (ISO). In GDS, OIDs are also used to identify object classes. When you create new attribute types, you are responsible for tagging them with new,

properly allocated OIDs. See your Directory Service administrator for OID assignments. In CDS, attribute types are identified by strings, which can be representations of OIDs.

- Name

A DCE name corresponds to an entry in some service participating in the DCE namespace, usually a directory service. (See “Structure of DCE Names.”)

- Global Name

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

- Local Name

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

- Access Control List

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

Objects in the GDS namespace have ACLs associated with them, but they are not DCE Security Service ACLs.

- Replication

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

- Caching

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

Structure of DCE Names

The following subsections describe the structure of the names found in a DCE environment. DCE names can consist of several different parts, which reflect the federated nature of the DCE namespace. (See “The Federated DCE Namespace” on page 9.) A DCE name has some combination of the following elements. They must occur in this order, but most elements are optional.

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

- Application name

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

In the discussion that follows, a DCE name *element* is the single piece of a name string enclosed between a consecutive pair of slashes. For example, in the following string, the substring `o=OSF` is an element, and so is `abc` and the entire name contains (counting the ... element) a total of seven elements.

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

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

DCE Name Prefixes

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

- /...* This is the *global root*. It signifies that the immediately following elements form the name of a global namespace entry. If the name is a DCE cell-name, the entry it represents contains binding information for that cell (more specifically, for a CDS server in that cell). A cell-name does not contain the CDS part of the entry (as described in the following subsections). The above example of a global entry, `/. . . /C=US/o=OSF/OU=DCE/hosts/abc/self` does not contain the binding information for the cell. The binding information for the cell is contained in `/. . . /C=US/o=OSF/OU=DCE`, and is the name of the cell.
- /.*: This is the *cell root*. It is an alias for the global prefix plus the name of the local cell; that is, the cell in which the prefix is being used. It signifies that the immediately following elements taken together form the name of a cell namespace entry.
- /:* This is the *filespace root*. It is an alias for the global prefix, the name of the local cell, and the DFS junction.

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

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

Names of Cells

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

GDS Cell Names: GDS elements always consist of a substring in which an abbreviation or acronym in capital letters is followed by an = (equal sign), followed by a string value. As you will learn in more detail in Chapter 2, “Programming in the CDS Namespace” on page 17. These substrings represent pairs of attribute types and attribute values.

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

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

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

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

In the above name, the single element *cs.univ.edu* is the cell name, that is, the cell's name in the DNS namespace. The DNS name consists of up to four domain names (depending on the name assigned to the cell), separated by dots.

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

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

CDS Names

Note: z/OS DCE supports all prefixes, GDS cell names or DNS cell names, CDS names, junctions, and application names.

After the cell name or cell root prefix, the next part of a DCE name is often a CDS name. For example, in the following name, the CDS part is */subsys/druecker/docs*:

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

Or in the following name, the CDS part is */subsys/printers/docs*:

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

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

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

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

GDS Names

Note: z/OS DCE does not support GDS names. The GDS naming information presented here is intended to increase your understanding of DCE name structure and concepts.

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

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

Junctions in DCE Names

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

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

Application Names

The part of a DCE name that occurs after a junction into a DCE application is the application name. Security is the only supported example.

Security names occur after the SEC junction. They are typeless and resemble HFS names. The parts of the name within the DCE Directory Service are resolved, and then the rest of the name is handled by the Security Service. The entry is contained in the Security registry database. In the following example, the Security part of the DCE name is **/principal/ziggy**:

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

The Federated DCE Namespace

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

Figure 1 on page 10 shows a typical DCE namespace and the different services in which names reside.

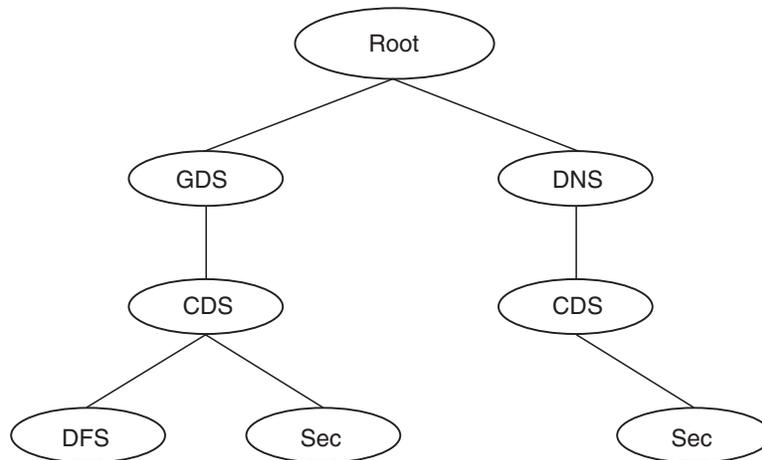


Figure 1. A Federated DCE Namespace

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

The GDS Namespace

This section is a brief overview of the main characteristics of the GDS namespace regarded separately from the XDS interface used to access it. More detailed information about GDS and XDS can be found in the section on GDS application programming beginning at Chapter 4, “GDS API: Concepts and Overview” on page 77, and the section on XDS/XOM supplemental information beginning at Chapter 10, “XDS Interface Description” on page 233, respectively.

In a GDS name such as `/.../C=US/O=OSF/OU=DCE`, the **C=US** and **O=OSF** elements do not refer to directory entries that are fundamentally different from the one represented by **OU=DCE**, unlike CDS or the hierarchical file system.

Thus, in the name string `/C=US/O=OSF/OU=DCE`, the element **C=US** refers to a one-level-down Country entry whose value is **US**, then to a two-levels-down Organization entry whose value is **OSF**, and then to a three-levels-down Organization Unit entry whose value is **DCE**. Concatenating these elements results in a valid path of entries from the directory root to the **DCE** entry. If the name of this entry is the cell name, the entry contains the binding information for the `/.../C=US/O=OSF/OU=DCE` cell. If not, it would not contain the binding information.

An Example GDS Namespace: Figure 2 on page 11 shows what a part of the DCE global namespace could look like. Levels in the tree of entries are numbered; the global root is at Level 0. The GDS structure rules as defined for DCE allow only country name entries at the next level under the root; organization name and locality name entries can exist at the level below a country name. An organizational unit name can be a child of an organizational name entry, and a common name can be a child of a locality name. The details of the GDS rules for the valid types and locations of entries in the directory tree can be found in the documentation of the host that offers the GDS service.

The object entry `/C=US/O=OSF/OU=DCE` belongs to the Organizational Unit class. One of the object’s values is the CDS server binding information that is used to reach the cell from other DCE cells.

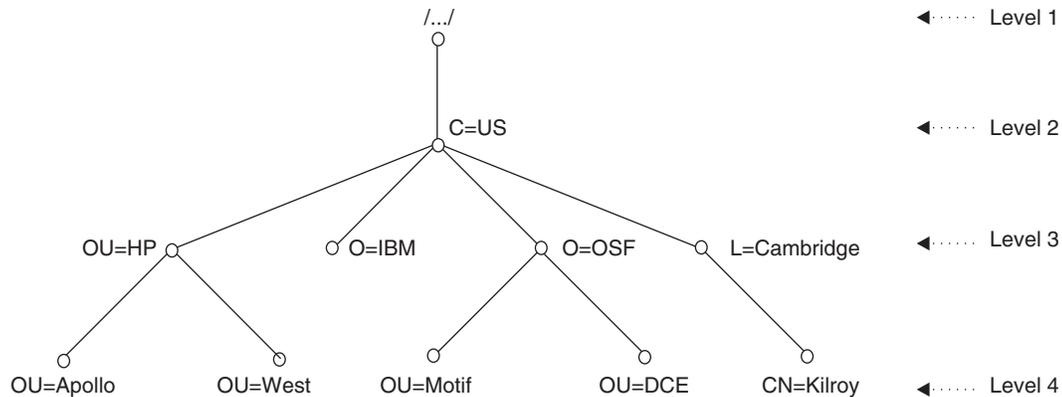


Figure 2. GDS Namespace Entries and Directory Objects

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

- The legal hierarchy of entries. What entries may be subordinate of other entries. These rules are what prevents, for example, countries from being subordinate to states.
- The allowable object classes, the mandatory and optional attributes of entries, and which attributes are the naming attributes.
- The allowable attribute types, associating a unique OID and an attribute syntax with each attribute type.
- The syntaxes of attributes that describe what attribute values look like, such as strings, numbers, or OIDs.

By installing the proper schema, an entry of any particular object class can have the two attributes needed to identify a cell. See the documentation for the host, in your cell, offering GDS services for a full description of how to set up a cell entry using either GDS or DNS.

The CDS Namespace

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

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

The cell namespace has a structure similar to the hierarchical file system. The CDS namespace is a tree of entries that grows from the root downward. The name entries are organized under directory entries, which can themselves be subentries of other directories. The cell root (represented by the prefix /.:) can

be thought of as the location you get when you dereference the cell's global name. New directories and new entries within the directories can be added anywhere in the tree, subject to the restrictions.

CDS Entries and CDS Attributes: There are three different kinds of CDS entries that are of significance to application programmers: object, soft link, and directory.

The object entries are the most primitive form. Data is stored in them. Directory entries contain other entries (that is, can have children) just like UNIX file system directories. Soft link entries are essentially alias names for other directory or object entries. Only object entries and soft links can be created by applications (using the XDS/XOM APIs); directories have to be created and manipulated with the **dcecp** or **cdscp** commands.

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

The term **attribute** is information of a particular type concerning an object and appearing in an entry that describes the object in the directory information database. It denotes the attribute's type and a sequence of one or more attribute values, each accompanied by an integer denoting the value's syntax. As applied to namespace entry objects, attribute has roughly the same meaning in CDS and GDS. The main difference is that CDS does not restrict or control the combinations of attributes attached to entries written in its namespace.

Other Namespaces

For information about names contained in the Security namespace, refer to the *z/OS DCE Application Development Guide: Core Components*.

Access to Objects in the Federated DCE Namespace

An application can access objects in the federated DCE namespace indirectly by attempting to reference objects in a foreign cell's DCE namespace.

For example, if an application attempts to locate RPC binding information from the following CDS entry:

```
../../../../C=US/O=IBM/OU=DEPTG71/servers/printers/prt1
```

and the application is located in a cell other than `../../../../C=US/O=IBM/OU=DEPTG71`, then the GDA daemon will look up the location of the foreign CDS server for the application.

LDAP support for the GDA daemon allows client applications to specify DCE cellnames that are in a typed (or X.500) format and use the GDA daemon to lookup this DCE cellname information in an external naming service which supports the Lightweight Directory Access Protocol (LDAP). Information about the LDAP programming interface is in the *z/OS SecureWay Security Server LDAP Client Programming*.

Programming Interfaces to the DCE Directory Service

There are two programming interfaces for accessing the DCE Directory Service.

The XDS Interface

The main programming interface to all services within the DCE Directory Service is XDS/XOM, as defined by X/Open. The calls correspond to the X.500 service requests, including Read, List (enumerate children), Search, Add Entry, Modify Entry, Modify RDN, and Remove Entry. XDS uses XOM to define and manipulate data structures (called *objects*) used as the parameters to these calls, and used to describe the directory entries manipulated by the calls. XOM is extremely flexible, but also somewhat complex. You use the interfaces in different ways, depending on the underlying directory service you are addressing. For example, CDS entry names are typeless, however the attribute used at the XDS interface is typed using a special type which indicates that it is typeless. GDS entry names are typed names. The XDS/XOM interfaces are used in an identical manner regardless of whether the name service used is GDS or CDS. The difference is that the name supplied in one case has a *typeless* portion to it. This difference is reflected in the use of the interface.

The RPC Name Service Interface

The DCE RPC facility supports an interface to the Directory Service that is specific to RPC and is layered on top of DCE Directory Service interfaces; it is called the Name Service Independent (NSI) interface. NSI can manipulate three object classes: entries, groups, and profiles, which were created to store RPC binding information. NSI data is stored in CDS. Programming using this interface is discussed in the *z/OS DCE Application Development Guide: Introduction and Style* and the *z/OS DCE Application Development Guide: Core Components*.

Namespace Junction Interfaces

For information about programming interfaces to names that occur in namespace junctions, see the documentation for that component. For example, for information about using Security names, see the *z/OS DCE Application Development Guide: Core Components*.

Part 2. CDS Application Programming

This part of the book describes DCE Directory Service application programming in the Cell Directory Service (CDS) namespace. It describes the contents of the CDS namespace, where applications should put their data, and what the valid CDS characters and names are. In addition, how to use the XDS programming interface to access data in the CDS namespace is explained.

Chapter 2. Programming in the CDS

Namespace	17
Initial Cell Namespace Organization	17
The Cell Profile	18
The LAN Profile	19
The CDS Clearinghouse	19
The Hosts Directory	19
The Subsystems Directory	19
The /: DFS Alias	20
DFS and Security Service Junctions	20
Recommended Use of the CDS Namespace	20
Storing Data in CDS Entries	20
Access Control for CDS Entries	23
Valid Characters and Naming Rules for CDS	25
Metacharacters	27
Additional Rules	27
Maximum Name Sizes	29
Use of Object Identifiers	31

Chapter 3. XDS and the DCE Cell

Namespace	33
Introduction to Accessing CDS with XDS	33
Using the Reference Material in this Chapter	33
What You Cannot Do with XDS	34

What Must Be Set Up	34
XDS Objects	34
Object Attributes	36
Interface Objects and Directory Objects	36
Directory Objects and Namespace Entries	38
Values That an Object Can Contain	39
Building a Name Object	39
A Complete Object	41
Class Hierarchy	42
Class Hierarchy and Object Structure	42
Public and Private Objects and XOM	42
XOM Objects and XDS Library Functions	43
Accessing CDS Using the XDS Step-by-Step Procedure	43
Reading and Writing Existing CDS Entry Attributes Using XDS	43
Creating New CDS Entry Attributes	55
Object-Handling Techniques	57
Using XOM to Access CDS	58
Dynamic Creation of Objects	59
XDS/CDS Object Recipes	60
Input XDS/CDS Object Recipes	60
Input Object Classes for XDS/CDS Operations	61
Attribute and Data Type Translation	72

Chapter 2. Programming in the CDS Namespace

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

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

For information on the details of accessing the CDS namespace through the XDS/XOM interface, see Chapter 3, “XDS and the DCE Cell Namespace” on page 33 and the section on XDS/XOM supplementary information beginning at Chapter 10, “XDS Interface Description” on page 233. For information about using XDS/XOM to access the GDS portion of the DCE namespace, see the section on GDS application programming beginning at Chapter 4, “GDS API: Concepts and Overview” on page 77.

Initial Cell Namespace Organization

The following subsections describe the organization of a cell’s namespace after it has initially been configured. For more information on configuring a cell, see *z/OS DCE Administration Guide*.

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

Figure 3 on page 18 shows what the cell namespace looks like after a cell has been configured and before any additional directories or entries have been added to it by system administrators or applications. In the figure, ovals represent directories, rectangles represent simple entries, circles represent soft links, and triangles represent namespace junctions.

All of the name entries shown in Figure 3 on page 18 are created for use with NSI routines; that is, they all contain server-binding information, and exist to enable clients of one sort or another to find servers. These are referred to as *RPC entries*.

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

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

Each of the entries is explained in detail in the following subsections. See *z/OS DCE Administration Guide* for detailed information on the contents of the initial DCE cell namespace.

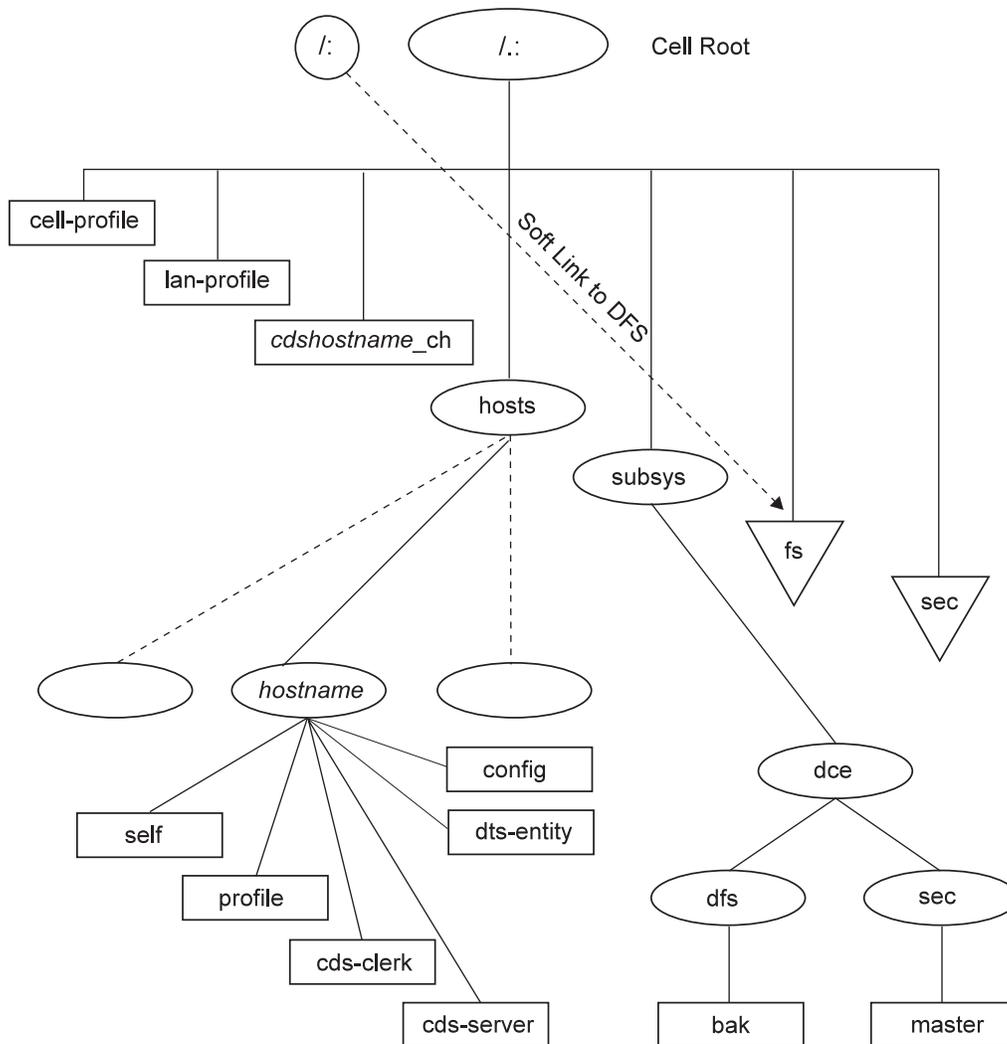


Figure 3. The Cell Namespace after Configuration

The Cell Profile

The **./cell-profile** entry is an RPC profile entry that contains the default list of namespace entries to be searched by clients trying to bind to certain basic services. An RPC profile is a class of namespace entry used by the RPC NSI routines. When a client imports bindings from such an entry, it imports, through the profile, from an ordered list of RPC entries containing appropriate bindings. The list of entries is keyed by their interface UUID, so that only bindings to servers offering the interface sought by the client are returned. The entries listed in the profile exist independently in the namespace, and can be accessed directly using the RPC NSI in the normal way rather than using a profile. The profile is simply a way of organizing clients' searches. For further information, see the *z/OS DCE Application Development Guide: Core Components*.

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

The LAN Profile

The `./lan-profile` is a LAN-oriented default list of namespace entries of services, that is used when relative positions of the servers in the network topography are of important to their prospective clients.

The CDS Clearinghouse

The `./cdshostname_ch` is the namespace entry for the clearinghouse `cdshostname`, where `cdshostname` is the name of the host machine on which a CDS server has been installed.

A clearinghouse is the database managed by a CDS server; CDS directory replicas are physically stored there. For more information about clearinghouses, see *z/OS DCE Administration Guide*. All clearinghouse namespace entries lie at the cell root, and there must be at least one in a DCE cell. Only `./cdshostname_ch` is listed here, because it is the minimum necessary for a configured cell.

The Hosts Directory

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

- self** This entry contains bindings to the host's RPC daemon (**rpcd**, also known as the *endpoint mapper*), which is responsible for dynamically resolving the partial bindings that it receives in incoming RPCs from clients attempting to reach servers resident on this host.
- profile** This entry is the default profile entry for the host. This profile contains in its list of entries at least the `./cell-profile` entry described in "The Cell Profile" on page 18.
- cds-clerk** This entry contains bindings to the host's resident CDS clerk.
- cds-server** This entry contains bindings to a CDS server.

The Subsystems Directory

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

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

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

The /: DFS Alias

The `/:` is created and set up as a soft link to the `/:/fs` entry, which is the DFS database junction. Note, however, that the name `/:` is well-known, whereas the name `/:/fs` is not. Thus using `/:` makes an application more portable. A CDS soft link entry is an alias to some other CDS entry. A soft link is created through the **dcecp** or **cdscp** commands. The procedure is described in *z/OS DCE Administration Guide*.

DFS and Security Service Junctions

The `/:/fs` entry is the Distributed File Service junction entry. This is the entry for a server that manages the DFS file location database.

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

Recommended Use of the CDS Namespace

CDS data is maintained in a loosely consistent manner. This means that when the writeable copy of a replicated name is updated, the read-only copies may not be updated for some period of time, and applications reading from those nonsynchronized copies can receive stale data. This contrasts with distributed databases, which use multiphase commit protocols that prevent readers from accessing potentially stale or inconsistent data while the write operations are being propagated to all copies of the data. You can specifically request data from the master copy, which is guaranteed to be up-to-date, but replication advantages are then lost. Only do so when it is important to obtain current data.

Note: This section is referring to the CDS clearinghouse, which includes the master and replicated copies of the CDS database. It does not refer to the CDS cache.

Storing Data in CDS Entries

Some CDS entries may contain either information that is immediately useful or meaningful to applications. Other entries may contain RPC information that enables application clients to reach application servers, that is, binding handles for servers, which are stored and retrieved using the NSI routines. In either case, the entry's name should be a meaningful identification label for the information that the entry contains. The reason is that the namespace entry names are the main clue that users and applications have to the available set of resources in the DCE cell. Using the CDS namespace to store and retrieve binding information for distributed applications is the function of DCE RPC NSI. See the *z/OS DCE Application Development Guide: Core Components* for information on that aspect of namespace usage.

In general, applications can store data into CDS object entry attributes in any XDS-expressible form. Table 5 on page 72 and Table 6 on page 72 contain XDS-to-CDS data type translations. If you add new attributes to the **cds_attributes** file, together with a meaningful CDS syntax (that is, data type identifier) and name, then the attribute is displayed by **dcecp show** operations and **cdscp show** commands when executed on CDS entries containing instances of that attribute.

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

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

You can create new subdirectories as long as you do not disturb the namespace's configured structure. Keep in mind that CDS directories must be created with the **dcecp** or **cdscp** commands; they cannot be directly created by applications. An application can be written, however, which calls the **dcecp** or **cdscp** commands to perform the action.

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

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

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

Or you could change the subtree structure so that: (1) new **entries** are added to hold the data, (2) the server bindings are still held in separate wholly RPC entries, and (3) each group of entries was located under a subdirectory named for the printer:

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

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

Figure 4 on page 22 illustrates some of the preceding suggestions, added to the initial configuration namespace structure shown in Figure 3 on page 18.

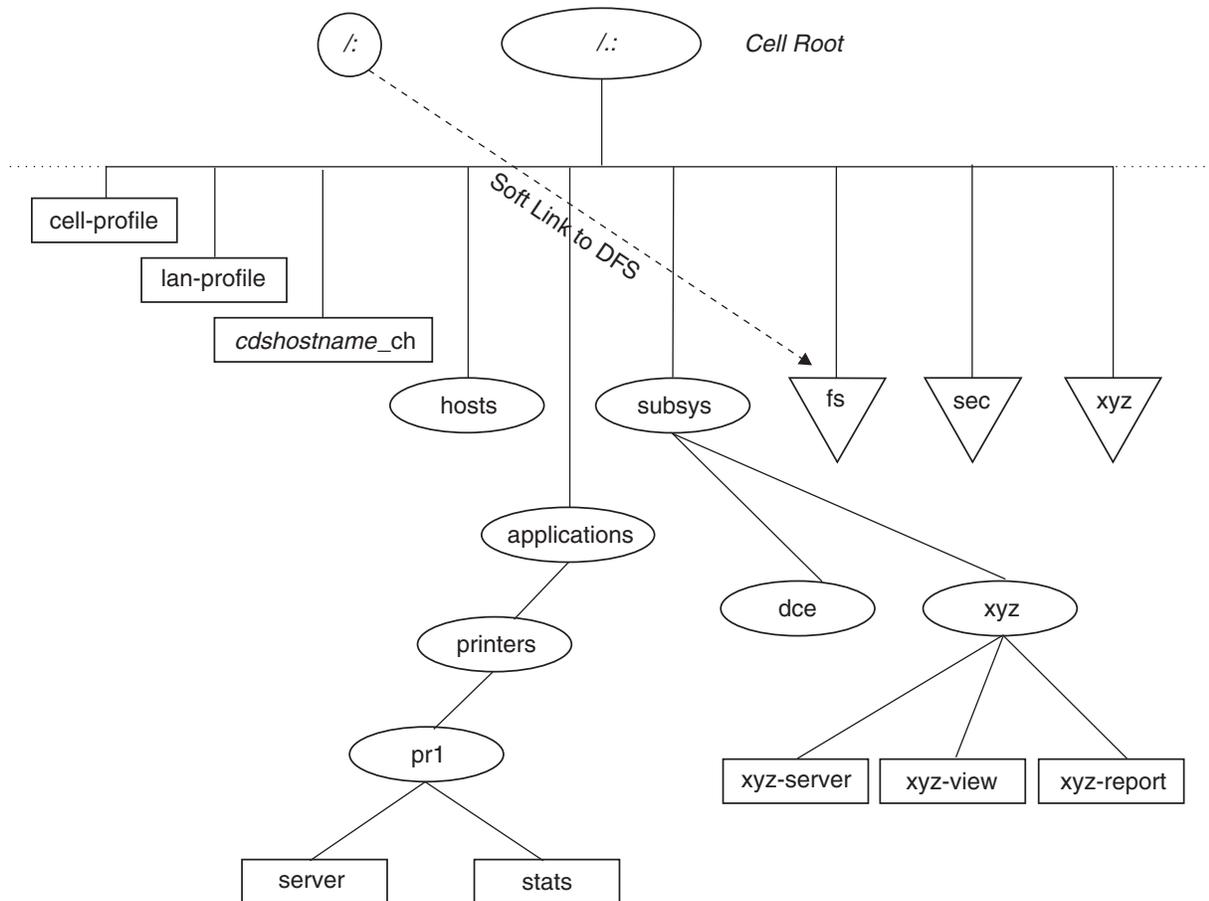


Figure 4. A Possible Namespace Structure

In Figure 4, the vendor of the **xyz** subsystem has set up an **xyz** directory under **./:subsys** in which the system's servers are exported. This cell also has an **./:applications** directory, in which the **printers** directory contains separate directories for each installed printer available on the system; the directory for **pr1** is illustrated in this figure. In the **pr1** directory, **server** is an RPC entry containing exported binding handles, and **stats** is an entry created and maintained through the XDS interface.

2. How should the entries be constructed?

Because CDS allows you to add as many attributes as you want to an object entry, it is up to you to impose some restraint in doing so. In view of the XDS overhead involved in reading and writing single CDS attributes, it makes sense to combine multiple related attributes under single entries (that is, in the same directory object) where they can be read and written in single calls to **ds_read()** or **ds_modify_entry()**. This way, for example, you only have to create one interface input object (to pass to **ds_read()**) to read all the attributes, which you can do with one call to **ds_read()**. You can then separate out the returned subobjects that you are interested in and ignore the rest. Chapter 3, "XDS and the DCE Cell Namespace" on page 33 contains detailed discussions of XDS programming techniques.

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

3. Should a directory or an entry be created?

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

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

Access Control for CDS Entries

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

DCE permission lists are called Access Control Lists (ACLs). ACLs are one of the features of the DCE Security Service used by the Cell Directory Service. ACLs are used to test a principal's *authorization* to access or manipulate objects. The authorization mechanism for all CDS entries is handled by CDS itself. Associated with each CDS entry is a list of DCE principals, and attached to each principal is a list of permissions, such as read, write, delete, and test. Users logged in as DCE principals must possess sufficient permissions to be able to manipulate a given CDS entry. CDS keeps this information on principals and their associated permissions, that is, the ACL, as attributes of the CDS entry. You can access and maintain this information using the `acl_edit` tool.

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

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

- | | |
|-------------------|--|
| Initial Container | Used to generate the initial ACL for any directories created within the directory. |
| Initial Object | Used to generate ACLs for entries created within the directory. |

Every CDS directory also has its own ACL, just like any other CDS object. This ACL is generated from the parent directory's Initial Container template when the child directory is created. The Initial Container template also serves as a template for the templates of child directories, and own Initial Container templates.

Manipulating ACLs: There are two ways to manipulate ACLs: either through the **acl_edit** command (which is documented in *z/OS DCE Command Reference*) or through the DCE ACL application interface, which consists of routines documented in *z/OS DCE Application Development Reference*. (These routines have names in the form of **sec_acl_...()**.)

Initializing ACLs: After creating a CDS directory using the **dcecp** or **cdscp** commands (in other words, they cannot be created by applications), you usually will run the **acl_edit** command to set up the new directory's ACLs the way you want them. (The new directory will have inherited its ACLs and its templates from the directory in which it was created, as explained in "Creating ACLs" on page 23.) In addition to modifying the directory's own ACL, you may want to modify its two templates. To edit the latter, you can specify the **-ic** argument (for the Initial Container template) or the **-io** argument (for the Initial Object template); otherwise, you will edit the object ACL.

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

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

An example of such an ambiguous name would be a CDS clearinghouse entry, such as the *cdshostname_ch* entry discussed earlier in "The CDS Clearinghouse" on page 19. With the **-e** option, you would edit the ACL on the namespace entry, as follows:

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

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

Similarly, there is a *bind_to_entry* parameter by which the caller of **sec_acl_bind()** can indicate whether the entry object's ACL is desired, or the ACL on the object for which the entry contains binding information.

Namespace ACLs at Cell Configuration: The ACLs attached to the CDS namespace at configuration are described in *z/OS DCE Configuring and Getting Started*. The following ACL permissions are defined for CDS objects. The single letter in parentheses for each item represents the DCE notation for that permission. These single letters are identical to the untokenized forms returned by **sec_acl_get_printstring()**.

- | | |
|-------------------|---|
| read (r) | Permits a principal to look up an object entry and view its attribute values |
| write (w) | Permits a principal to change an object's modifiable attributes, except for its ACLs |
| insert (i) | Permits a principal to create new entries in a CDS directory. It is used with directory entries only. |
| delete (d) | Permits a principal to delete a name entry from the namespace. |

- test (t)** Permits a principal to test whether an attribute of an object has a particular value, but does not permit it to actually see any of the attribute values. In other words, read permission for the object is not granted. The **test** permission allows an application to verify the value of a particular CDS attribute without reading it.
- control (c)** Permits a principal to modify the entries in the object's ACL. The control permission is automatically granted to the creator of a CDS object.
- administer (a)** Permits a principal to issue **dcecp** and **cdscp** commands that control the replication of directories. It is used with directory entries only.

Detailed instructions on the mechanics of setting up ACLs on CDS objects can be found in the *z/OS DCE Administration Guide*.

For CDS directories, **read** and **test** permissions are sufficient for ordinary principals to access the directory and to read (and test) the entries there. Principals that you want to be able to add entries in a CDS directory should have **insert** permission for that directory. Entries created by the RPC NSI routines (for example, when a server exports bindings for the first time) are automatically set up with the correct permissions. However, if you are creating new CDS directories for NSI use, you should be sure to give prospective user principals **insert** permission to the directory, so that servers can create entries when they export their bindings. A general list of the permissions required for the various RPC NSI operations can be found in the *z/OS DCE Application Development Reference*. Information on the separate RPC NSI routines (all of whose names are in the form **rpc_ns_...()**) can be found in the *z/OS DCE Application Development Reference*, where it describes the permissions required for the described operation.

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

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

Although it may not have **read** access to the **utilities** directory itself.

Valid Characters and Naming Rules for CDS

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

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

Figure 5 on page 26 details the valid characters in CDS names, and the valid character GDS and DNS names as used by CDS interfaces.

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

Key: Valid in CDS,GDS, and DNS names
 Valid only in CDS and GDS names
 Valid only in CDS names

Figure 5. Valid Characters in CDS, GDS, and DNS Names

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

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

Metacharacters

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

Table 1. Metacharacters and Their Meaning

Directory Service	Character	Meaning
CDS	/	Separates elements of a name (simple names)
CDS	*	When used in the rightmost simple name of a name entered in a CDSCP show or list command acts as a wild card, matching 0 or more characters
CDS	?	When used in the rightmost simple name of a name entered in a CDSCP show or list command acts as a wild card, matching exactly one character
CDS	\	Used where necessary in front of a \ (backslash), an * (asterisk), or a ? (question mark) to escape the character (indicates that the following character is not a metacharacter)
GDS	/	Separates relative distinguished names (RDNs)
GDS	,	Separates multiple attribute type-value pairs (attribute value assertions) within an RDN
GDS	=	Separates an attribute type and value in an attribute value assertion
GDS	\	Used in front of / (slash), a ,(comma), or = (equal sign) to escape the character (indicates that the following character is not a metacharacter).
DNS	.	Separates elements of a name

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

Additional Rules

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

Table 2 (Page 1 of 2). Summary of CDS, GDS, and DNS Characteristics

Characteristic	CDS	GDS	DNS
Character set	[a to z] [A to Z] [0 to 9] plus space and special characters shown in Figure 5 on page 26	[a to z] [A to Z] [0 to 9] plus . : , ' + - = () ? / and space	[a to z] [A to Z] [0 to 9] plus . and -
Metacharacters	/ * ? \	/ , = \	.

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

Characteristic	CDS	GDS	DNS
Restrictions	<p>Simple names cannot contain a / (slash).</p> <p>The first simple name following the global cell name (or /.: prefix) cannot contain an = (equal sign).</p> <p>When entering a name as part of a CDSCP show or list command, you must use a \ (backslash) to escape any * (asterisk) or ? (question mark) character in the rightmost simple name. Otherwise, the character is interpreted as a wild card.</p>	<p>Relative distinguished names cannot begin or end with a / (slash).</p> <p>Attribute types must begin with an alphabetic character, can contain only alphanumerics, and cannot contain spaces¹.</p> <p>You must use a \ (backslash) to escape a / (slash), a , (comma), and an = (equal sign) when using them as anything other than metacharacters.</p> <p>Multiple, consecutive, unescaped occurrences of / (slashes), , (commas), = (equal signs) and \ (backslashes) are not allowed.</p> <p>Each attribute value assertion contains exactly one unescaped = (equal sign).</p>	<p>The first character must be alphabetic.</p> <p>The first and last characters cannot be a . (dot) or - (dash).</p> <p>Cell names in DNS must contain at least one . (dot); they must be more than one level deep.</p>
Case-matching rules	Case exact	Attribute types are matched case-insensitive. The case-matching rule for an attribute value can be case-exact or case-insensitive, depending on the rule defined for its type at the DSA.	Case insensitive
Internal Representation	Case exact	Depends on the case-matching rule defined at DSA. If rule says case insensitive, alphabetic characters are converted to all lowercase. Spaces are removed regardless of the case-matching rule.	Alphabetic characters are converted to all lowercase characters.
Ordering of name elements.	Big-endian (left to right from root to lower-level names).	Big-endian (left to right from root to lower-level names).	Little-endian (right to left from root to lower-level names).

Notes:

1. An alternative method of specifying attribute types is by object identifier, a sequence of digits separated by dots (.).

Maximum Name Sizes

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

Table 3. Maximum Sizes of Directory Service Names

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

Valid Characters for GDS Naming Attributes: This section describes the valid character sets for the GDS naming attributes. Although GDS is not supported on z/OS DCE, you need to understand this information because a cell can have a name which is partly in GDS. To construct the global name to the entry in CDS, you should know the rules for the GDS portion.

The values of the country attributes are restricted to the ISO 3166 Alpha-2 code representation of country names. (For more information, see *z/OS DCE Administration Guide*.)

The character set for all other naming attributes is the T61 graphical character set. It is described in “T61 Syntax” on page 30.

T61 Syntax: Figure 6 shows the T61 graphical character set.

Note: The 1) entry in the table indicates that you should not use the codes in column 2/row 3 and column 2/row 4. Instead, use the appropriate code in column A.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0			SP	0	@	P		p				°			Ω	κ
1			!	1	A	Q	a	q			i	+ ₋	`		Æ	æ
2			"	2	B	R	b	r			∅	2	'			đ
3			1)	3	C	S	c	s			£	3			a ₋	
4			1)	4	D	T	d	t			\$	x			Ĥ	ĥ
5			%	5	E	U	e	u			¥	μ	-			l
6			&	6	F	V	f	v			#	¶	˘		IJ	ij
7			'	7	G	W	g	w			§	•	•		L•	l•
8			(8	H	X	h	x				÷	••		Ł	ł
9)	9	I	Y	i	y							∅	ø
A			*	:	J	Z	j	z					◦		Œ	œ
B			+	;	K	[k				<<	>>	˘			β
C			,	<	L		l	l				1/4	—			
D			-	=	M]	m					1/2	"		ƒ	ƒ
E			.	>	N		n					3/4	˘		η	η
F			/	?	O	_	o					˘	v		'n	

Figure 6. T61 Syntax

The administration interface supports only characters smaller than 0x7e for names. The **XDS** Application Programming Interface (API) supports the full T61 range as indicated in Figure 6.

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

Only certain combinations of diacritical characters and basic letters are valid. They are shown in Figure 7 on page 31.

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

Figure 7. Combinations of Diacritical Characters and Basic Letters

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

Use of Object Identifiers

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

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

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

For example, the OID block 1.3.22 was allocated to OSF by ISO. OSF can now generate, for example, the OID 1.3.22.1.1.4 and allocate it to identify some DCE directory object. (The OID **1.3.22.1.1.4** identifies the RPC profile entry object attribute.)

OSF is responsible for making sure that **1.3.22.1.1.4** is not used to identify any other attribute. Thus, as long as all OIDs are generated only from within each owner's properly obtained block, and as long as

each block owner makes sure that the OIDs generated within its block are properly used, each OID will always be a universally valid identifier for its associated value.

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

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

```
1.3.22.1.1.4      RPC_Profile      byte
```

This enables users to see **RPC_Profile** in output, rather than the meaningless string **1.3.22.1.1.4**. Further details about the **cds_attributes** file and OIDs can be found in the *z/OS DCE Administration Guide*.

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

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

The *name* is a guaranteed-unique series of values for a global directory entry name. If the directory is GDS, the *name* is a series of type/value pairs, such as:

```
C=US 0=0SF
```

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

Chapter 3. XDS and the DCE Cell Namespace

This chapter describes how you use the XDS programming interface to access the CDS namespace. Use the XDS interface if you are interested in accessing the namespace for more than RPC information. If you are not interested in such information, skip this section.

- “Introduction to Accessing CDS with XDS,” is an introduction to using XDS in the CDS namespace.
- “XDS Objects” on page 34 describes XDS objects and how they are used to access CDS data.
- “Accessing CDS Using the XDS Step-by-Step Procedure” on page 43 is a step-by-step procedure for writing an XDS program to access CDS.
- “Object-Handling Techniques” on page 57 contains examples of using the XOM interface to manipulate objects.
- “XDS/CDS Object Recipes” on page 60 contains details of the structure of XDS/CDS objects.
- Finally, “Attribute and Data Type Translation” on page 72 contains translation tables between XDS and CDS for attributes and data types.

Introduction to Accessing CDS with XDS

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

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

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

Using the Reference Material in this Chapter

Complete descriptions of all the XDS and XOM functions used in CDS operations can be found in the *z/OS DCE Application Development Reference*, which you should have beside you as you read through the examples in this chapter. Definitive descriptions of all XDS and XOM class types can be found in the XDS/XOM supplementary material section beginning with Chapter 10, “XDS Interface Description” on page 233. In particular, refer to “XDS Errors” on page 241 for information about XDS error objects, which are not discussed in this chapter.

Complete descriptions for some of the objects required as *input* parameters by XDS functions when accessing a CDS namespace can be found in “XDS/CDS Object Recipes” on page 60. Abbreviated definitions for these same objects can be found with all the others in the supplementary XDS/XOM material beginning at Chapter 10, “XDS Interface Description” on page 233. XOM functions are

general-purpose utility routines that operate on objects of any class, and take the rest of their input in conventional form.

Slightly less detailed descriptions of the *output* objects you can expect to receive when accessing CDS through XDS are also given in “XDS/CDS Object Recipes” on page 60. You do not have to construct objects of these classes yourself; you just have to know their general structure so that you can disassemble them using XOM routines.

No information is given in this chapter about the **DS_status** error objects that are returned by unsuccessful XDS functions; a description of all the subclasses of **DS_status** requires a chapter in itself. Code for a rudimentary general-purpose **DS_status**-handling routine can be found in “The teldir.c Program” on page 170.

What You Cannot Do with XDS

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

- You cannot create or modify directory entries; the **ds_modify_rdn()** function does not work in a CDS namespace. These operations must be performed through the **dcecp** or **cdscp** commands. For more information, see the *z/OS DCE Command Reference*.
- You cannot perform XDS searches on the cell namespace; the XDS function **ds_search()** does not work. The reason is that the CDS has no concept of a hierarchy of entry attributes, such as the X.500 schema. The **ds_compare()** function, however, does work.

What Must Be Set Up

If you are planning to use XDS to access the cell namespace in a one-cell environment (that is, your cell does not need to communicate with other DCE cells), you do not need to set up a cell entry in GDS for your cell because the XDS functions simply call the appropriate statically linked CDS routines to access the cell namespace. In these circumstances, XDS always tries to access the local cell when given an untyped (non-X.500) name.

For XDS to be able to access any nonlocal cell namespace, that cell must be registered (that is, have an entry) in the global namespace.

For information on setting up your cell name, see the *z/OS DCE Administration Guide*.

XDS Objects

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

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

XDS is such an interface. For the most part, XDS functions only accept or return values as objects. The objects themselves are indeed always the same data type. Namely, pointers to arrays of *object descriptor* (C **struct**) elements. Contained within these **OM_descriptor** element structures are unions that can actually accommodate all the different kinds of values an object can be called on to hold. To allow the interface to make sense of the unions, each **OM_descriptor** also contains a **syntax** field, which indicates the data type of that descriptor's union. There is also a record of what the descriptor's value actually is, for example, whether it is a name, a number, an address, a list, and so on. This information is held in the descriptor's **type** field.

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

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

Figure 8 illustrates one such object descriptor.

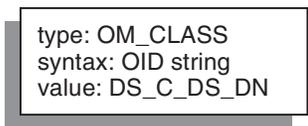


Figure 8. One Object Descriptor

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

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

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

Figure 9 illustrates the arrangement of a complete object descriptor array.

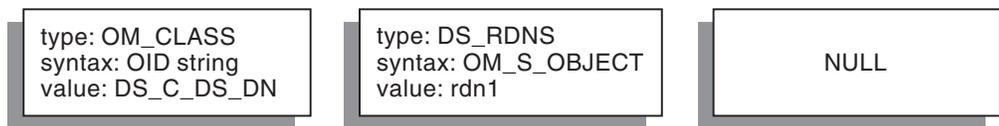


Figure 9. A Complete Object Represented

Object Attributes

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

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

Figure 10 shows a three-layer compound object: the top-level superobject, **dn_object**, contains the subobject **rdn1**, which in turn contains the subobject **ava1**.

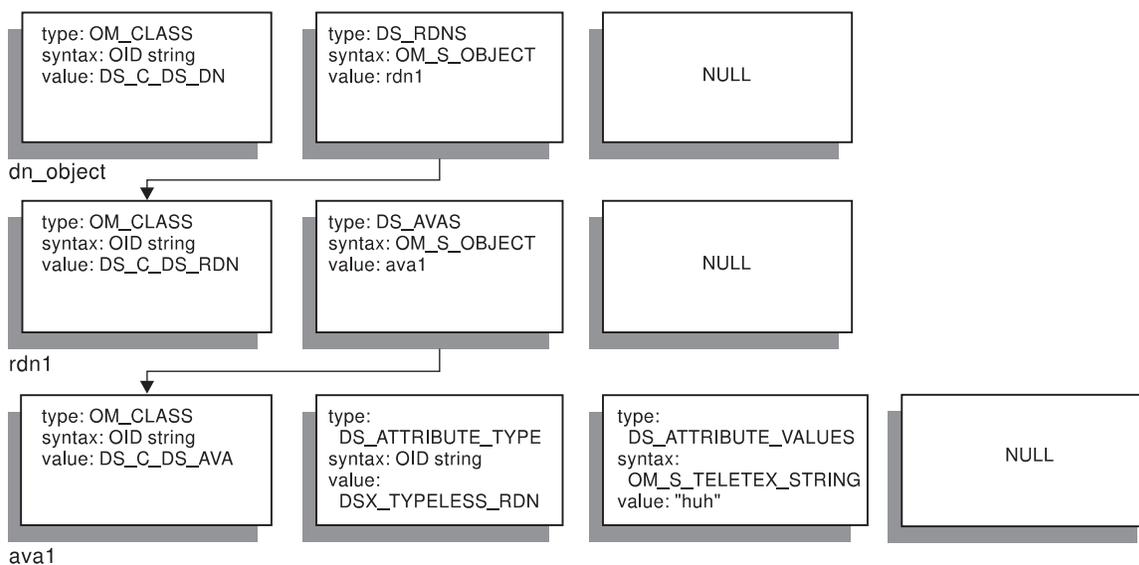


Figure 10. A Three-Layer Compound Object

Interface Objects and Directory Objects

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

Programmers work with XDS objects. XDS objects have explicit data representations that can be directly manipulated with programming language operators. Some of these techniques are described in this chapter; others can be found in Chapter 7, "Example Application Programs" on page 159.

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

Figure 11 on page 37 illustrates the relationship between XDS (also called *interface*) objects and directory objects. The figure shows an application passing several properly initialized XDS objects to some XDS

function; it then takes some action, which affects the attribute contents of certain directory objects. The application never works with the directory objects; it works with the XDS interface objects.

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

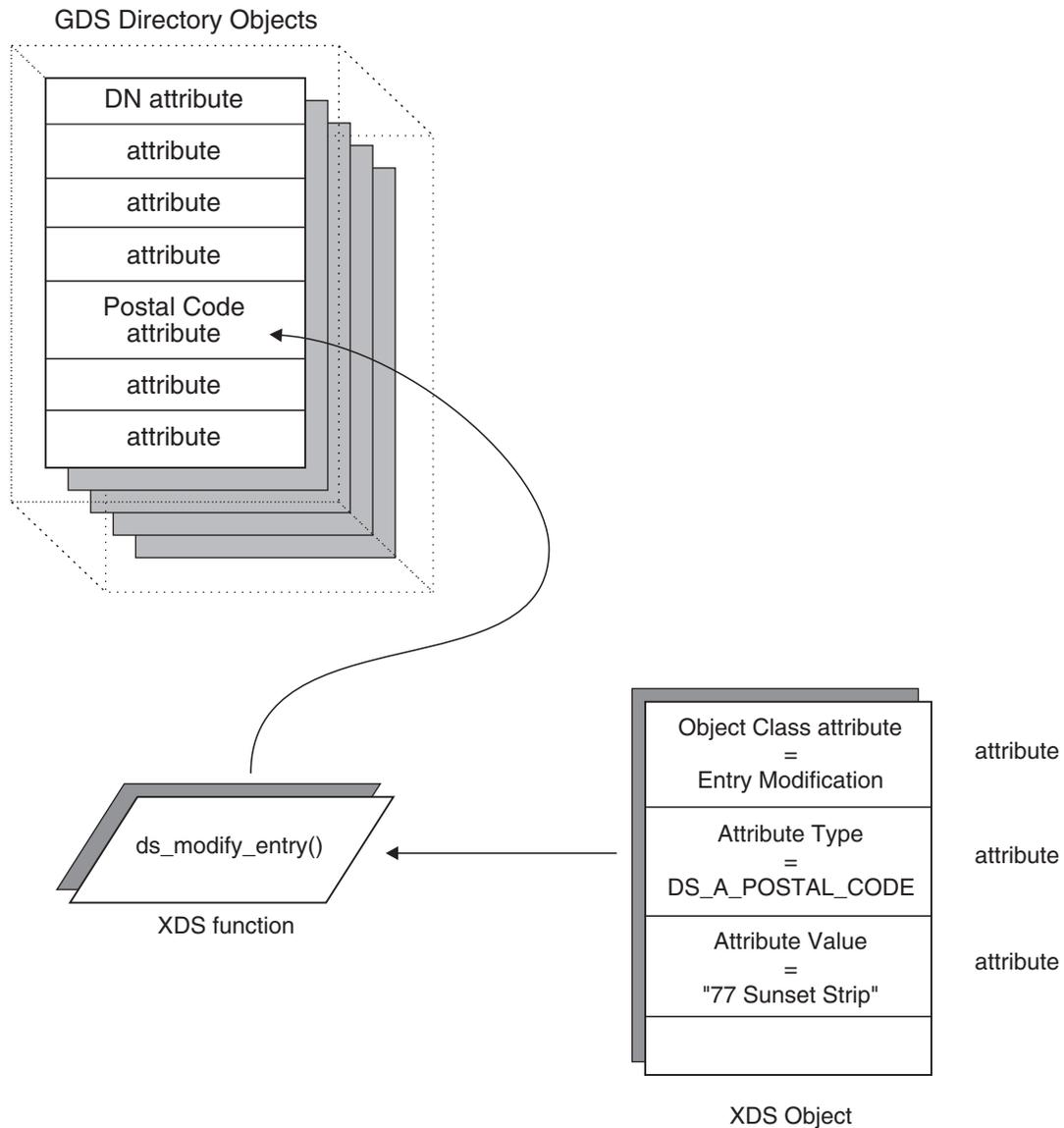


Figure 11. Directory Objects and XDS Interface Objects

Directory Objects and Namespace Entries

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

Figure 12 shows the relationship of entry names in the GDS namespace to the GDS directory objects to which they refer.

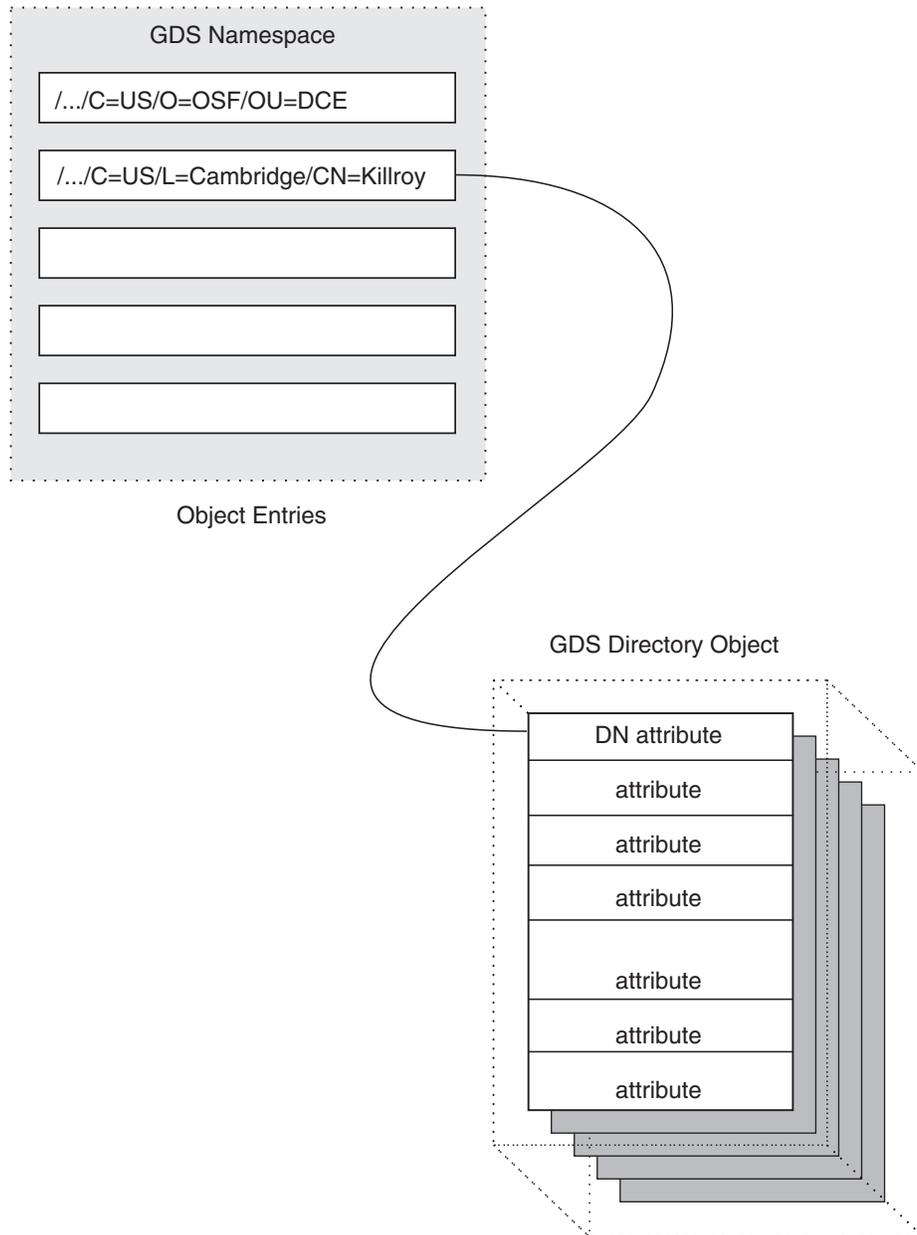


Figure 12. Directory Objects and Namespace Entries

Values That an Object Can Contain

There are many different classes of objects defined for the XDS interface; still more are defined by the X.500 standard for general directory use. But only a small number of classes are needed for XDS/CDS operations, and only those classes are discussed in this chapter. Information about other classes can be found in the GDS application programming section beginning at Chapter 4, “GDS API: Concepts and Overview” on page 77.

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

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

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

Building a Name Object

Complete definitions for all the object classes required as input for XDS functions can be found in “XDS/CDS Object Recipes” on page 60. Among them is the class for distinguished name objects, called **DS_C_DS_DN**. There you will learn that this class of object has two attributes: its class attribute, which identifies it as a **DS_C_DS_DN** object, and a second attribute, which occurs multiple times in the object. Each instance of this attribute contains as its value one piece of the full name; for example, the directory name **hosts**.

The **DS_C_DS_DN** attribute that holds the entry name piece, or Relative Distinguished Name, is defined by the class rules to hold, not a string, but another object of the Relative Distinguished Name class (**DS_C_DS_RDN**).

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

```

static OM_descriptor    Full_Entry_Name_Object[] = {

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

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

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

    OM_NULL_DESCRIPTOR
/* ~~~~~ */
/* Macro to fill a descriptor with proper */
/* null values. */

};

```

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

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

The definition for **DS_C_DS_RDN** can be found in "The DS_C_DS_RDN Object" on page 66. This class object is defined, like **DS_C_DS_DN**, to have only one attribute (with the exception of the **OM_Object** attribute, which is mandatory for all objects). The one attribute, **DS_AVAS**, holds the value of one relative name. The syntax of this value is **OM_S_OBJECT**, meaning that the value of **DS_AVAS** is a pointer to yet another object descriptor array:

```

static OM_descriptor    Country_RDN[] = {

    OM_OID_DESC(OM_CLASS, DS_C_DS_RDN),

    {DS_AVAS, OM_S_OBJECT, {0, Country_Value}},
/* ~~~~~ */
/* ~~~~~ */
    OM_NULL_DESCRIPTOR
};

```

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

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

The rules for the **DS_C_AVA** class can be found in “The DS_C_AVA Object” on page 66 following **DS_C_DS_RDN**. They tell us that **DS_C_AVA** objects have two attributes aside from the omnipresent **OM_Object**; namely:

- **DS_ATTRIBUTE_VALUES**

This attribute holds the object’s value.

- **DS_ATTRIBUTE_TYPE**

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

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

```
static OM_descriptor    Country_Value[] = {  
  
    OM_OID_DESC(OM_CLASS, DS_C_AVA),  
  
    OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_COUNTRY_NAME),  
  
    {DS_ATTRIBUTE_VALUES, OM_S_PRINTABLE_STRING, OM_S_LOCAL_STRING, OM_STRING("US")},  
/*                                     */  
/*                                     Macro to properly */  
/*      fill the "value" union with the null-terminated string. */  
  
    OM_NULL_DESCRIPTOR  
};
```

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

A Complete Object

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

Normally, you do not statically declare objects in real applications. The steps outlined in the preceding text are given as a method for determining what an object looks like. Once you have done that, you can then write routines to create the objects dynamically. An example of how to do this can be found in the **teldir.c** example application in “The teldir.c Program” on page 170.

The process of object building is fairly easy. There are only five different object classes needed for input to XDS functions when accessing CDS, and only one of those, the **DS_C_DS_DN** class, has more than

one level of subobjects. The rules for all five of these classes can be found in the GDS chapters of this book. To use the GDS references, you should know a few things about class hierarchy.

Class Hierarchy

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

XDS/XOM classes are seldom nested more than two or, at most, three layers. All inherited attributes are explicitly listed in the object descriptions that follow, so you do not have to worry about class hierarchies here. However, the complete descriptions of XDS/XOM objects in the XDS/XOM supplementary information section later in this book rely on statements of class inheritance to fill out their attribute lists for the different classes. Refer to this section, beginning in Chapter 10, “XDS Interface Description” on page 233 for information about the classes of objects that can be returned by XDS calls to handle those returned objects.

Class Hierarchy and Object Structure

Note that class hierarchy is different from object structure. Object structure is the layering of object arrays that was previously described in the **DS_C_DS_DN** declaration in “Building a Name Object” on page 39. It occurs when one object contains another object as the value of one or more of its attributes.

For recursive objects, one object can point to another object as one of its attribute values. The layering of subobjects below superobjects in this way is described in “XDS/CDS Object Recipes” on page 60.

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

Public and Private Objects and XOM

In “Building a Name Object” on page 39, you saw how a multilevel XDS object can be statically declared in C code. Now imagine that you have written an application that contains such a static **DS_C_DS_DN** object declaration. From the point of view of your application, that object is nothing but a series of arrays, and you can manipulate them with all the normal programming operators, just as you can any other data type. Nevertheless, the object is syntactically acceptable to any XDS (or XOM) function that is prepared to receive a **DS_C_DS_DN** object.

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

These two kinds of objects are the same as far as their classes and attributes are concerned. The only difference between them is in the way they are accessed. The public objects that an application explicitly creates or declares in its own memory area are just as accessible as any of the other data storage it uses. However, private objects are created and held in the system memory of the XDS interface. Applications

get handles to private objects, and in order to access the private objects' contents, they have to pass the handles to Object Management functions. The Object Management (XOM) functions make up a sort of all-purpose companion interface to XDS. While XDS functions typically require some specific class object as input, the XOM functions accept objects of any class and perform useful operations on them.

If a private object needs to be manipulated, one of the XOM functions, **om_get()**, can be called to make a public copy of the private object. Private objects unlike public objects do not have to be explicitly operated on; instead, you can access them through the XOM interface and let it do most of the work. You still have to know something about the logical representation of objects, however, to use XOM.

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

XOM Objects and XDS Library Functions

To call an XDS library function, do the following:

1. Decide what input parameters you must supply to the function.
2. Bundle up these parameters in objects (that is, arrays of object descriptors) in an XDS format.

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

With XDS, you have to perform a lot of call parameter management, but the interface is easy to use. The dependence of the XDS functions on objects makes them easy to call, once the objects are correctly set up.

Accessing CDS Using the XDS Step-by-Step Procedure

The following subsections provide a walk-through of the steps of some typical XDS/CDS operations. They describe what is involved in using XDS to access existing CDS attributes and how you can create and access new CDS entry attributes.

Reading and Writing Existing CDS Entry Attributes Using XDS

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

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

The **./hosts/hostname/self** entry, which is created at the time of cell configuration, contains binding information for the machine *hostname*. This is a simple RPC NSI entry and as an example provides a simple demonstration.

Following are the header inclusions and general data declarations.

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

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

The XDS/XOM interface defines numerous object identifier string constants, which are used to identify the many object classes, parts, and pieces (among other things) that it needs to know about. To make sure that these OID constants do not collide with any other constants, the interface refers to them with the string **OMP_O_** prefixed to the user-visible form. For example, **DS_C_DS_DN** becomes **OMP_O_DS_C_DS_DN** internally. To make application instances consistent with the internal form, use **OM_EXPORT** to import *all* XDS-defined or XOM-defined OID constants used in your application.

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

OM_EXPORT( DS_C_ATTRIBUTE )
OM_EXPORT( DS_C_ATTRIBUTE_LIST )
OM_EXPORT( DS_C_AVA )
OM_EXPORT( DS_C_DS_DN )
OM_EXPORT( DS_C_DS_RDN )
OM_EXPORT( DS_C_ENTRY_INFO_SELECTION )
OM_EXPORT( DSX_TYPELESS_RDN )
    /* ...Special OID for an untyped (i.e., non-X.500) "Relative */
    /* Distinguished Name". Defined in xdscds.h header.          */
```

A further important effect of **OM_EXPORT** is that it builds an **OM_string** structure to hold the exported Object Identifier hexadecimal string. As explained in Chapter 2, "Programming in the CDS Namespace" on page 17, OIDs are not numeric values, but strings. Comparisons and similar operations on OIDs must access them as strings. Once an OID has been exported, you can access it using its declared name. For example, the hexadecimal string representation of **DS_C_ATTRIBUTE** is contained in **DS_C_ATTRIBUTE.elements**, and the length of this string is contained in **DS_C_ATTRIBUTE.length**.

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

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

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

```
osf.org.dce
```

Then the entire string **osf.org.dce** would be held in a single object whose **DS_ATTRIBUTE_TYPE** would be **DSX_TYPELESS_RDN**.

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

name cannot be resolved. If the untyped portion of the name is at the beginning, as would be the case with the name:

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

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

Static Declarations: The following are the static declarations you need:

```
/******  
/* Here are the objects that contain the string values for each */  
/* part of the CDS entry's global name... */  
  
static OM_descriptor Country_String_Object[] = {  
    OM_OID_DESC(OM_CLASS, DS_C_AVA),  
    OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_COUNTRY_NAME),  
    {DS_ATTRIBUTE_VALUES, OM_S_PRINTABLE_STRING|OM_S_LOCAL_STRING, OM_STRING("US")},  
    OM_NULL_DESCRIPTOR  
};  
  
static OM_descriptor Organization_String_Object[] = {  
    OM_OID_DESC(OM_CLASS, DS_C_AVA),  
    OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_ORG_NAME),  
    {DS_ATTRIBUTE_VALUES, OM_S_PRINTABLE_STRING|OM_S_LOCAL_STRING, OM_STRING("OSF")},  
    OM_NULL_DESCRIPTOR  
};  
  
static OM_descriptor Org_Unit_String_Object[] = {  
    OM_OID_DESC(OM_CLASS, DS_C_AVA),  
    OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_ORG_UNIT_NAME),  
    {DS_ATTRIBUTE_VALUES, OM_S_PRINTABLE_STRING|OM_S_LOCAL_STRING, OM_STRING("DCE")},  
    OM_NULL_DESCRIPTOR  
};  
  
static OM_descriptor Hosts_Dir_String_Object[] = {  
    OM_OID_DESC(OM_CLASS, DS_C_AVA),  
    OM_OID_DESC(DS_ATTRIBUTE_TYPE, DSX_TYPELESS_RDN),  
    {DS_ATTRIBUTE_VALUES, OM_S_PRINTABLE_STRING|OM_S_LOCAL_STRING, OM_STRING("hosts")},  
    OM_NULL_DESCRIPTOR  
};  
  
static OM_descriptor Tamburlaine_Dir_String_Object[] = {  
    OM_OID_DESC(OM_CLASS, DS_C_AVA),  
    OM_OID_DESC(DS_ATTRIBUTE_TYPE, DSX_TYPELESS_RDN),  
    {DS_ATTRIBUTE_VALUES, OM_S_PRINTABLE_STRING|OM_S_LOCAL_STRING, OM_STRING("tamburlaine")},  
    OM_NULL_DESCRIPTOR }  
;  
  
static OM_descriptor Self_Entry_String_Object[] = {  
    OM_OID_DESC(OM_CLASS, DS_C_AVA),  
    OM_OID_DESC(DS_ATTRIBUTE_TYPE, DSX_TYPELESS_RDN),
```

```

{DS_ATTRIBUTE_VALUES, OM_S_PRINTABLE_STRING|OM_S_LOCAL_STRING, OM_STRING("self")},
OM_NULL_DESCRIPTOR
};

```

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

```

/*****
/* Here are the "Relative Distinguished Name" objects.          */

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

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

More information on GDS (X.500) names is provided in "X.500 Naming Concepts" on page 83. That information will clarify the need for the name complexity with RDNs and AVAs.

```

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

```

Other Necessary Objects for ds_read(): The `ds_read()` procedure takes requests in the form of a `DS_C_ENTRY_INFO_SELECTION` class object. However, if you refer to the recipe for this object class in “XDS/CDS Object Recipes” on page 60, you will find that it is much simpler than the name object; it contains no subobjects, and its declaration is straightforward.

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

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

```

static OM_descriptor      Entry_Info_Select_Object[] = {

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

```

Miscellaneous Declarations: The following are declarations for miscellaneous variables:

```

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

DS_feature featureList[] = {

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

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

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

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

OM_sint      dummy;

```

```

    /* ...for unsupported ds_read() argument          */
OM_private_object      readResultObject;
    /* ...to receive entry information read from CDS by "ds_read()" */

OM_type I_want_entry_object[] = {DS_ENTRY, OM_NO_MORE_TYPES};
OM_type I_want_attribute_list[] = {DS_ATTRIBUTES, OM_NO_MORE_TYPES};
OM_type I_want_attribute_value[] = {DS_ATTRIBUTE_VALUES, OM_NO_MORE_TYPES};
    /* ...arrays to pass to "om_get()" to extract subobjects */
    /* from the result object returned by "ds_read()"          */

OM_value_position number_of_descriptors;
    /* ...to hold number of attribute descriptors returned    */
    /* by "om_get()"                                           */

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

```

The Main Program: This section describes the main program. Three calls usually precede any use of XDS.

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

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

The service packages are specified next. Packages consist of groups of objects, together with the associated supporting interface functionality, designed to be used for some specific end. For example, to access the (X.500) Global Directory, specify **DSX_GDS_PKG**. This example uses the basic XDS service, so **DS_BASIC_DIR_CONTENTS_PKG** is specified. The *featureList* parameter to **ds_version()** is an array, not an object, because objects (which depend on the set of packages supported) are not being handled yet:

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

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

The third necessary call is to **ds_bind()**. This call brings up the directory service, which binds to a Directory System Agent (DSA), the GDS server, through a Directory User Agent (DUA), the GDS client. The **DS_DEFAULT_SESSION** parameter calls for a default session. The alternative is to build and fill out your own **DS_C_SESSION** object, specifying such things as DSA addresses, and pass that. The default is used in this example:

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

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

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

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

If the call succeeds, the private object `readResultObject` contains a series of `DS_C_ATTRIBUTE` subobjects, one for each attribute read from the cell name entry. More information for the `DS_C_READ_RESULT` object can be found in Chapter 11, “XDS Class Definitions” on page 241, but the following is an outline of the object’s structure:

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

    DS_C_ENTRY_INFO
        DS_FROM_ENTRY: OM_S_BOOLEAN
        DS_OBJECT_NAME: object(DS_C_NAME)
        DS_ATTRIBUTES: one or more object(DS_C_ATTRIBUTE)

    DS_C_NAME == DS_C_DS_DN
        DS_RDNS: object(DS_C_DS_RDN)

        DS_C_DS_RDN
            DS_AVAS: object(DS_C_AVA)

            DS_C_AVA
                DS_ATTRIBUTE_TYPE: OID string
                DS_ATTRIBUTE_VALUES: anything

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

    DS_C_ATTRIBUTE
        DS_ATTRIBUTE_TYPE: OID string
        DS_ATTRIBUTE_VALUES: anything
```

Figure 13 on page 50 illustrates the general object structure of a `DS_C_READ_RESULT`, showing only the object-valued attributes, and only one `DS_C_ATTRIBUTE` subobject.

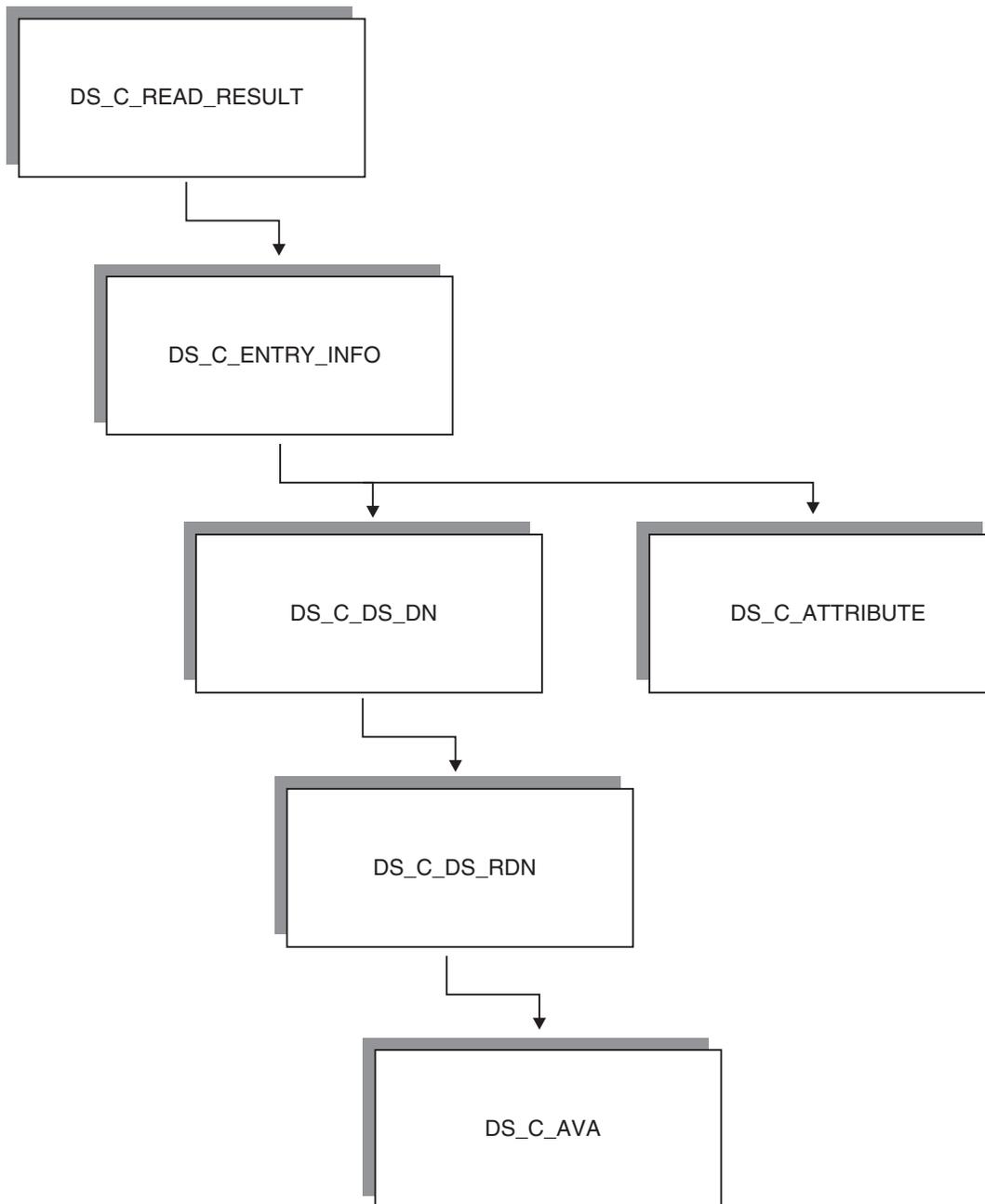


Figure 13. *DS_C_READ_RESULT* Object Structure

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

The **om_get()** function takes as its third input parameter an **OM_type_list**, which is an array of **OM_type**. Possible parameters are **DS_ENTRY**, **DS_ATTRIBUTES**, **DS_ATTRIBUTE_VALUES**, and anything that

can legitimately appear in an object descriptor's **type** field. The types specified in this parameter are interpreted according to the options specified in the preceding parameter. For example, the relevant attribute from the read result is **DS_ENTRY**. It contains the **DS_C_ENTRY_INFO** object, which in turn contains the **DS_C_ATTRIBUTE** objects. The **DS_C_ATTRIBUTE** objects contain the data read from the cell directory name entry. Therefore, you should specify the **OM_EXCLUDE_ALL_BUT_THESE_TYPES** option, which has the effect of excluding everything but the contents of the object's **DS_ENTRY** type attribute.

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

The following is the first call:

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

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

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

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

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

```

DS_C_ENTRY_INFO
  DS_FROM_ENTRY: OM_S_BOOLEAN
  DS_OBJECT_NAME: object(DS_C_NAME)
  DS_ATTRIBUTES: object(DS_C_ATTRIBUTE)
                    DS_C_ATTRIBUTE
                      DS_ATTRIBUTE_TYPE: OID string
                      DS_ATTRIBUTE_VALUES: anything

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

```

The italics indicate private subobjects. Figure 14 shows the **DS_C_ENTRY_INFO** object. Only one instance of a **DS_C_ATTRIBUTE** subobject is shown in the figure; usually there are several such subobjects, all at the same level, each containing information about one of the attributes read from the entry. These subobjects are represented in **DS_C_ENTRY_INFO** as a series of descriptors of type **DS_ATTRIBUTES**, each of which has as its value a separate **DS_C_ATTRIBUTE** subobject.

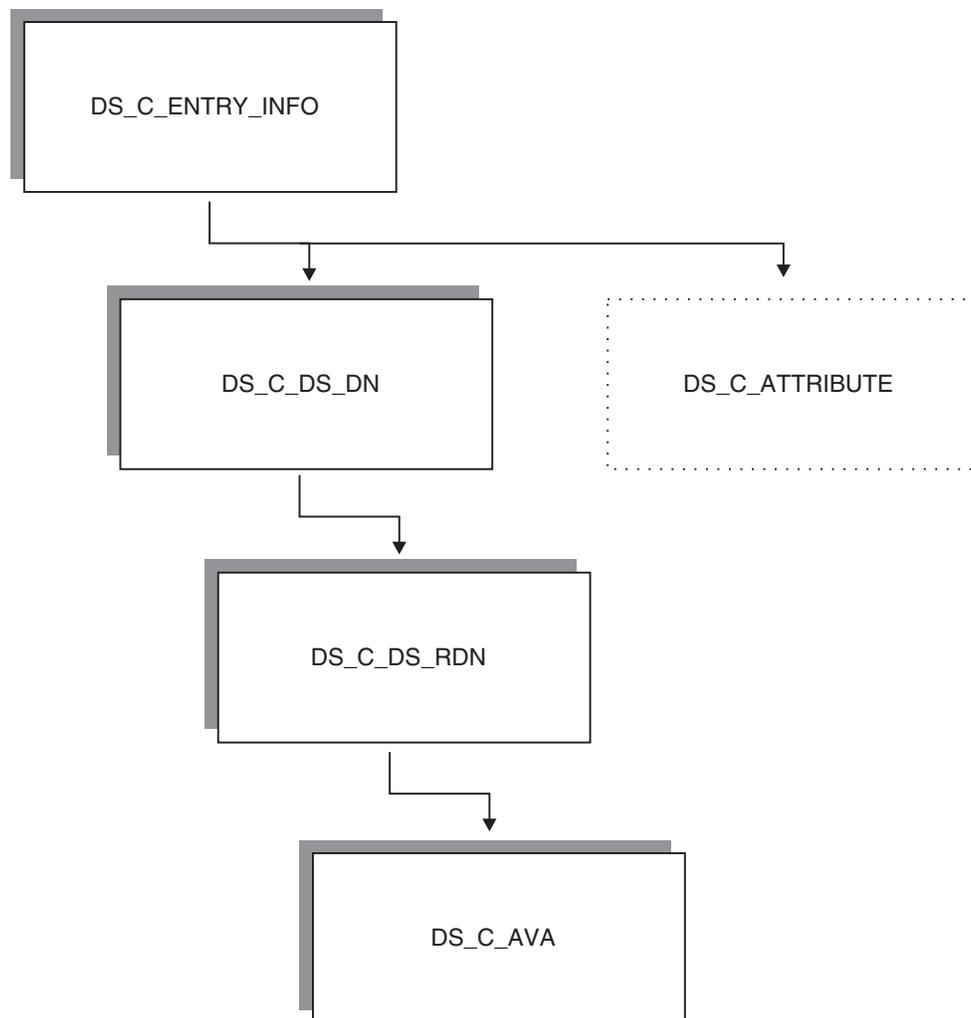


Figure 14. *DS_C_ENTRY_INFO* Object Structure

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

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

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

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

Representation of Object Values: The following is the layout of the **object** field in a descriptor's **value** union:

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

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

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

The following is the layout of the descriptor itself:

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

Thus, if **entry** is a pointer to the **DS_C_ENTRY_INFO** object, then **entry->value.object.object** is the handle to the **DS_C_ATTRIBUTE** private object that you want next.

Extracting an Attribute Value: The last call yielded one separate **DS_C_ATTRIBUTE** subobject from the original returned result object:

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

Figure 15 illustrates what is left.

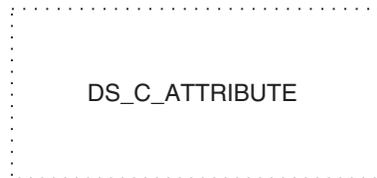


Figure 15. *DS_C_ATTRIBUTE* Object Structure

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

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

At this point, the value of **entry** is the base address of an object descriptor whose **entry->type** is **DS_ATTRIBUTE_VALUES**. Depending on the value found in **entry->syntax**, the value of the attribute can be read from **entry->value.string**, **entry->value.integer**, **entry->value.boolean**, or **entry->value.enumeration**. For example, suppose the value of **entry->syntax** is **OM_S_OCTET_STRING**. The attribute value, represented as an octet string (*not* terminated by a **NULL**), is found in **entry->value.string.elements**; its length is found in **entry->value.string.length**.

Note: The C declaration of the **OM_string** data type used for a data value of string syntax is:

```
typedef OM_uint32 OM_string_length;
typedef struct {
    OM_string_length length;
    void *elements;
} OM_string;
```

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

```
dcecp object show ././hosts/tamburlaine/self
```

or by using the **cdscp** command and entering:

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

For further information on **dcecp** and **cdscp**, see the *z/OS DCE Command Reference*.

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

```
/* make a public copy of ENTIRE object.* */  
  
omStatus = om_get(readResultObject, OM_NO_EXCLUSIONS,  
                ((OM_type_list) 0), OM_TRUE, ((OM_value_position) 0),  
                ((OM_value_position) 0),  
                &entry,  
                &number_of_descriptors);
```

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

```
/* delete service-generated public or private objects.**/  
  
omStatus = om_delete(readResultObject);  
omStatus = om_delete(entry);  
  
/* unbind from the GDS... */  
dsStatus = ds_unbind(session);  
  
/* close down the workspace... */  
dsStatus = ds_shutdown(xdsWorkspace);  
  
exit();
```

Creating New CDS Entry Attributes

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

Creating New Attributes: To create new attributes of your own in cell namespace object entries, you must:

1. Allocate a new ISO Object Identifier (OID) for the new attribute. For information on how to do this, see Chapter 2, "Programming in the CDS Namespace" on page 17 and the *z/OS DCE Administration Guide*.
2. Enter the new attribute's name and OID in the **cds_attributes** file, found in **cds_attributes**. This text file contains OID-to-readable string mappings that are used, for example, by the CDS administration command **cdscp** when it displays CDS entry attributes. Each entry also gives a syntax for reading the information in the entry itself. This should be congruent with the format of the data you intend to write in the attribute. For more information about the **cds_attributes** file, see the *z/OS DCE Administration Guide*.
3. In a C language header file, define an appropriate OID string constant to represent the new attribute.

For example, the following shows the **xds.cds.h** definition for the CDS **CDS_Class** attribute:

```
#define OMP_0_DSX_A_CDS_Class    "\\x2B\\x16\\x01\\x03\\x0F"
```

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

```
#define OMP_O_DSX_A_CDS_Brave_New_Attrib "your_OID"
```

4. In an application, call the XDS **ds_modify_entry()** routine to add the attribute to the cell namespace entry of your choice.

Coding Examples: For the following code fragments, a set of declarations similar to those in the previous examples is assumed.

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

```
static OM_descriptor    Entry_Modification_Object_1[] = {
    OM_OID_DESC(OM_CLASS, DS_C_ENTRY_MOD),
    OM_OID_DESC(DS_ATTRIBUTE_TYPE, DSX_A_CDS_Brave_New_Attrib),
    {DS_ATTRIBUTE_VALUES, OM_S_PRINTABLE_STRING|OM_S_LOCAL_STRING, OM_STRING("0 brave new attribute")},
    {DS_MOD_TYPE, OM_S_ENUMERATION, DS_ADD_ATTRIBUTE},
    OM_NULL_DESCRIPTOR
};
```

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

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

A full description of this object can be found in “XDS/CDS Object Recipes” on page 60. There could be any number of additional attribute changes in the list; this would mean additional **DS_C_ENTRY_MOD** objects declared, and an additional **DS_CHANGES** descriptor declared and initialized in the **DS_C_ENTRY_MOD_LIST** object.

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

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

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

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

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

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

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

The **ds_add_entry()** function also requires a **DS_C_DS_DN** class object containing the new entry's full name. In the following example, every member of the name exists except for the last one, **my_book**:

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

Assuming that **Full_Entry_Name_Object** is a **DS_C_DS_DN** object, the following example code shows what the call looks like:

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

Object-Handling Techniques

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

Using XOM to Access CDS

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

The following technique is used to initialize the modification object:

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

The following new declarations are necessary:

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

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

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

char *my_string = "0 brave new attribute";
    /* ...value I want to write into attribute            */

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

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

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

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

```
omStatus = om_put(newAttributeMod_priv, /* Private object to copy */
                 /* attributes into */
                 OM_REPLACE_ALL, /* Which attributes to replace in */
                 /* destination object */
                 Entry_Modification_Object_2, /* Source object to copy */
                 /* attributes from */
                 types_to_include, /* List of attribute types we want */
                 /* copied */
                 0, 0); /* Start-stop index for multivalued attri- */
                 /* butes; ignored with OM_REPLACE_ALL */
```

Because `om_put()` ignores the class of the source object (the object from which attributes are being copied), it is not necessary to declare class descriptors for the source objects. In other words, the static

declarations could have omitted the **OM_CLASS** initializations if this technique were being used, for example:

```
static OM_descriptor      Entry_Modification_Object_2[] = {
/*      OM_OID_DESC(OM_CLASS, DS_C_ENTRY_MOD),          */
/*      Not needed for "om_put()"...                    */

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

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

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

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

Now make the whole thing public again:

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

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

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

Dynamic Creation of Objects

Objects can be completely dynamically allocated and initialized; however, you have to implement the routines to do this yourself. The examples in this section are code fragments; for complete examples, see Chapter 7, "Example Application Programs" on page 159.

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

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

When generating objects, use **malloc()** or **rpc_ss_allocate()** to allocate space for the number of objects desired, and then use macros (or functions) such as the preceding one to initialize the descriptors. The

following code fragment shows how this can be done for the top-level object of a **DS_C_DS_DN**, such as the one described near the beginning of this chapter. Recall that the **DS_C_DS_DN** has a separate **DS_RDNS** descriptor for each name piece in the full name.

```

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

/* Allocate space for that many descriptors, plus one for the      */
/* object class at the front, and a NULL descriptor at the      */
/* back...                                                       */
Name_Object = (OM_object)malloc((numberOfPieces + 2) * sizeof(OM_descriptor));
if(Name_Object == NULL)                                         /* "malloc()" failed */
return OM_MEMORY_INSUFFICIENT;

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

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

```

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

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

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

XDS/CDS Object Recipes

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

Input XDS/CDS Object Recipes

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

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

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

Table 4 on page 61 shows XDS functions and the objects given to them as input parameters.

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

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

Table 4. Directory Service Functions with their Required Input Objects

Function	Input Object
<code>ds_add_entry()</code>	<code>DS_C_NAME</code> <code>DS_C_ATTRIBUTE_LIST</code>
<code>ds_bind()</code>	None
<code>ds_compare()</code>	<code>DS_C_NAME</code> <code>DS_C_AVA</code>
<code>ds_initialize()</code>	None
<code>ds_list()</code>	<code>DS_C_NAME</code>
<code>ds_modify_entry()</code>	<code>DS_C_NAME</code> <code>DS_C_ENTRY_MOD_LIST</code>
<code>ds_read()</code>	<code>DS_C_NAME</code> <code>DS_C_ENTRY_INFO_SELECTION</code>
<code>ds_remove_entry()</code>	<code>DS_C_NAME</code>
<code>ds_shutdown()</code>	None
<code>ds_unbind()</code>	None
<code>ds_version()</code>	None

Input Object Classes for XDS/CDS Operations

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

Objects that are returned to you by the interface are not difficult to manipulate because the `om_get()` function allows you to go through them and retrieve only the value parts you are interested in and discard the parts of data structures you are not interested in. Some examples of how to do this are given in “Extracting the Data from the Read Result” on page 131. However, any objects you are required to supply as *input* to an XDS or XOM function are another matter: you must build and initialize these object structures yourself.

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

An object’s properties, such as what sort of values it can hold, how many of them it can hold, and so on, are determined by the *class* the object belongs to. Each class consists of one or more *attributes* that an object can have. The attributes hold whatever values the object contains. Thus, the objects are data structures that all look the same (and can be handled in the same way) from the outside, but whose specific data fields are determined by the class each object belongs to. At the abstract level, objects consist of attributes, just as structures consist of fields.

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

- **DS_C_ATTRIBUTE_LIST**
 - **DS_C_ATTRIBUTE**
- **DS_C_DS_DN**
 - **DS_C_DS_RDN**
 - **DS_C_AVA**
- **DS_C_ENTRY_MOD_LIST**
 - **DS_C_ENTRY_MOD**
 - **DS_C_ATTRIBUTE**
- **DS_C_ENTRY_INFO_SELECTION**

Note: The above list does **not** show the object class hierarchy; rather, it shows the object inclusions of subobjects by superobjects.

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

The illustrations in the following sections can be compared to the same object classes' tabular definitions in Chapter 11, "XDS Class Definitions" on page 241.

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

Its general structure is as follows:

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

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

Figure 16 on page 63 shows the attributes of the **DS_C_ATTRIBUTE_LIST** object.

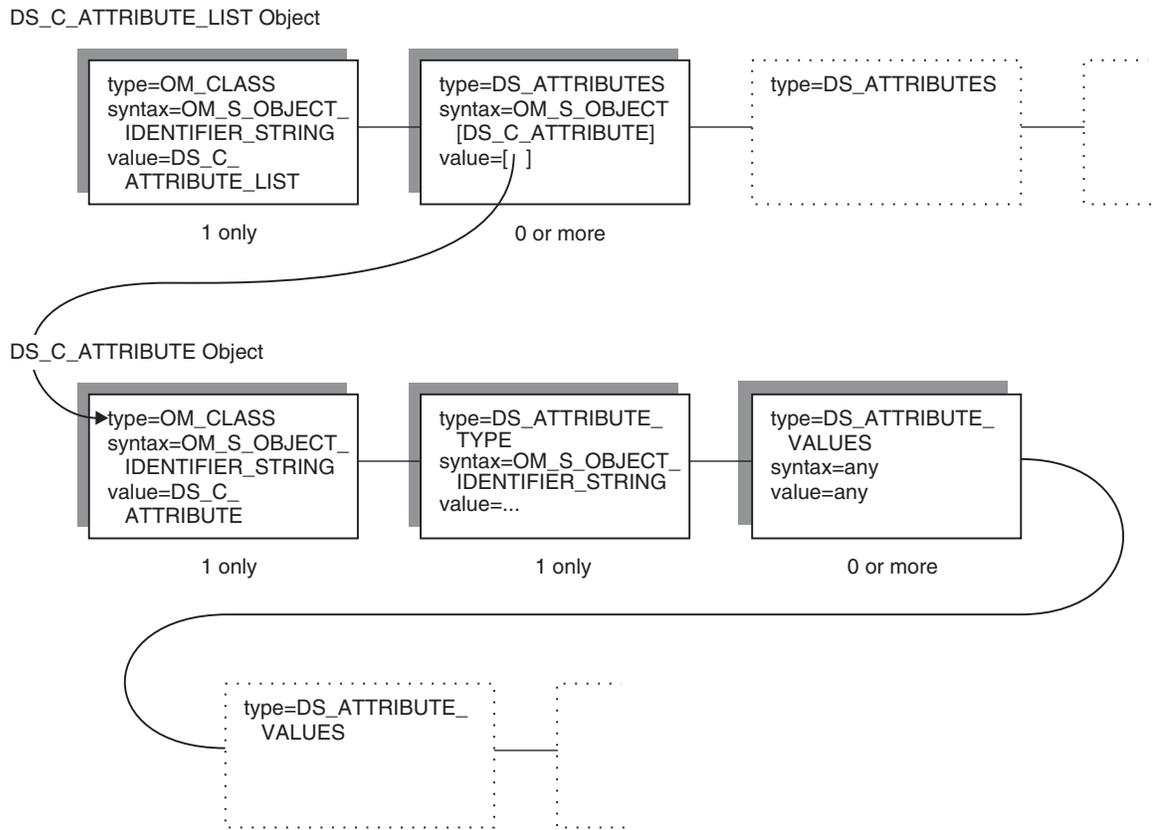


Figure 16. DS_C_ATTRIBUTE_LIST Object

- **OM_CLASS**

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

- **DS_ATTRIBUTES**

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

The DS_C_ATTRIBUTE Object: An object of this class can be an attribute of a **DS_C_ATTRIBUTE_LIST** object.

- **OM_CLASS**

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

- **DS_ATTRIBUTE_TYPE**

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

- **DS_ATTRIBUTE_VALUES**

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

Example Definition of a DS_C_ATTRIBUTE_LIST Object: The following code fragment is a definition of a **DS_C_ATTRIBUTE_LIST** object.

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

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

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

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

Figure 17 on page 65 shows the attributes of a **DS_C_DS_DN** object.

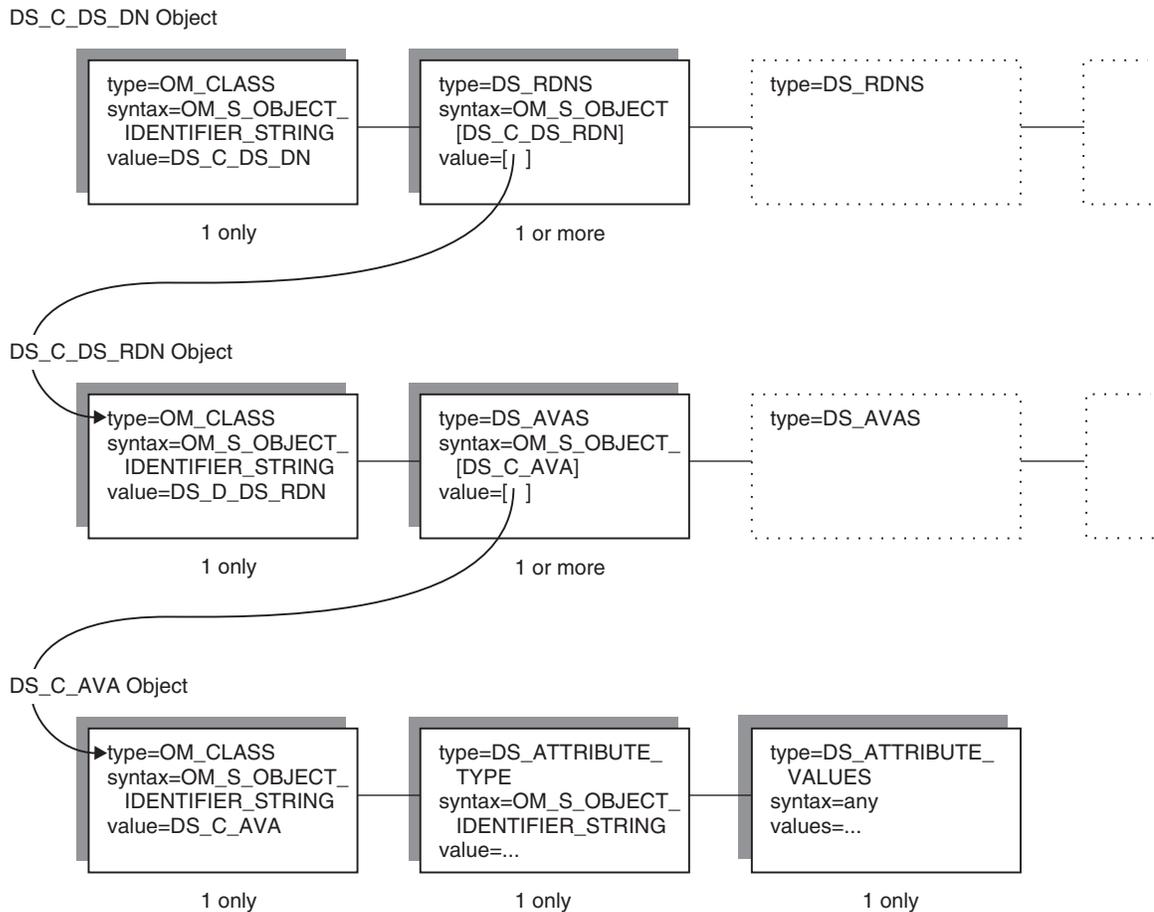


Figure 17. DS_C_DS_DN Object

- **OM_CLASS**

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

- **DS_RDNS**

This is an attribute whose value is a pointer to an object of class **DS_C_DS_RDN** (see "The DS_C_DS_RDN Object" on page 66). The **DS_C_DS_RDN** object is defined by a separate array of object descriptors whose base address is the value of the **DS_RDNS** attribute.

There are as many **DS_RDNS** attributes in a **DS_C_DS_DN** object as there are separate name components in the full directory entry name. For example, a total of six instances of the **DS_RDNS** attribute are required in the **DS_C_DS_DN** object to represent the following CDS entry name:

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

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

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

The DS_C_DS_RDN Object: **DS_C_DS_RDN** class objects are required as values for the **DS_RDNS** attributes of **DS_C_DS_DN** objects. **RDN** refers to the X.500 term Relative Distinguished Name that is used to signify a part of a full entry name. Separate objects of this class are not usually required as input to XDS functions.

The standard permits multiple AVAs in an RDN, but the DCE Directory and XDS API restrict an RDN to one AVA. GDS and XDS actually support multi-valued RDNs but CDS (at least through XDS) does not.

- **OM_CLASS**

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

- **DS_AVAS**

This is an attribute whose value is a pointer to an object of class **DS_C_AVA**. The **DS_C_AVA** object is defined by a separate array of object descriptors whose base address is the value of the **DS_AVAS** attribute.

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

The DS_C_AVA Object: The **DS_C_AVA** class object is used to hold an actual value. The value is usually in the form of one of the many different XOM string types. (For an illustration of its structure, see Figure 17 on page 65.) AVA refers to the X.500 term Attribute Value Assertion.

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

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

- **OM_CLASS**

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

- **DS_ATTRIBUTE_TYPE**

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

- **DS_ATTRIBUTE_VALUES**

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

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

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

For example, suppose you wanted to compare a certain value with a CDS entry's **CDS_Class** attribute's value. The identifiers for all the valid CDS entry attributes are found in the **cds_attributes** file in **cds_attributes**. The value of **DS_ATTRIBUTE_TYPE** would be **DSX_A_CDS_Class**, which is the label of an object identifier string, and **DS_ATTRIBUTE_VALUES** would contain some desired value, in the correct syntax for **CDS_Class**. The syntax also is found in the **cds_attributes** file; for **CDS_Class** it is **byte**; that is, a character string.

Example Definition of a DS_C_DS_DN Object: The following code fragment shows an example definition for a **DS_C_DS_DN** object.

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

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

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

The DS_C_ENTRY_MOD_LIST Object: **DS_C_ENTRY_MOD_LIST** class objects, which contain a list of changes to be made to some directory entry, must be passed to **ds_modify_entry()**. **DS_C_ENTRY_MOD_LIST** objects have the attributes shown in Figure 18 on page 68.

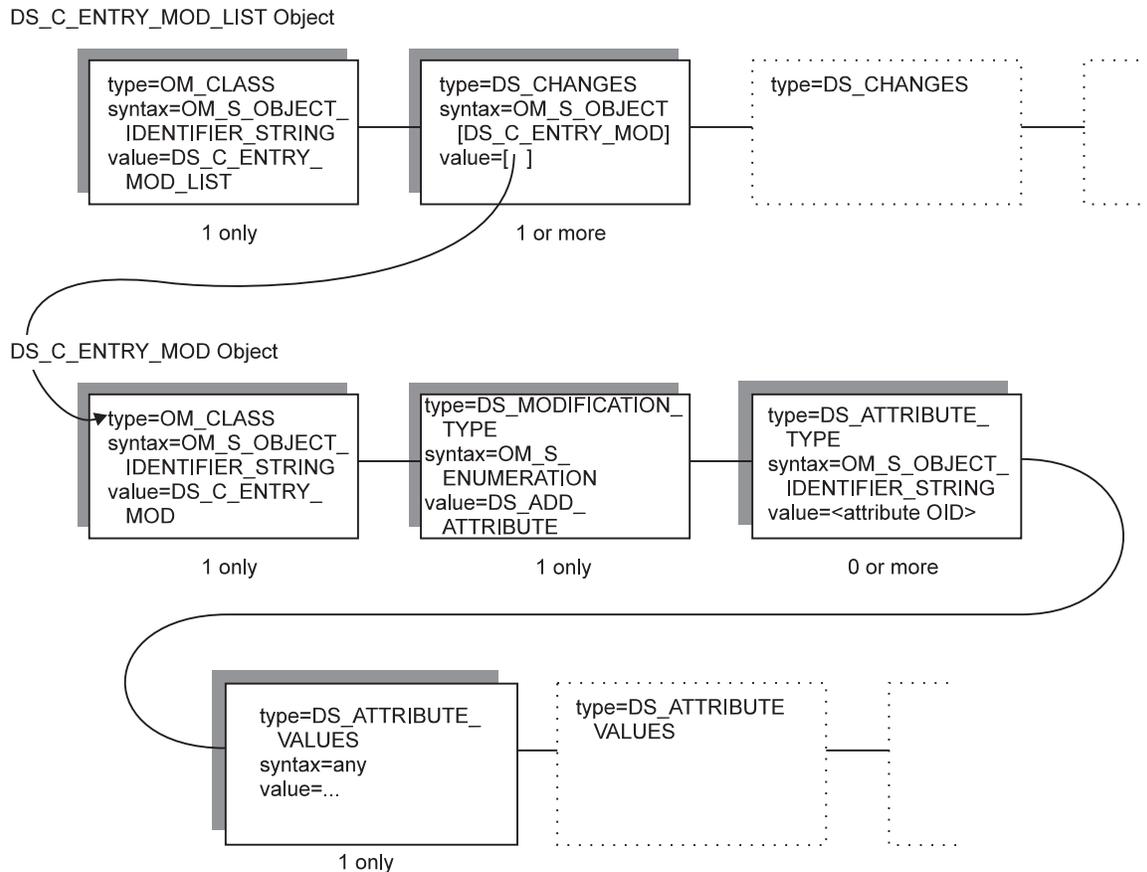


Figure 18. DS_C_ENTRY_MOD_LIST Object

- **OM_CLASS**

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

- **DS_CHANGES**

This is an attribute whose value is a pointer to an object of class **DS_C_ENTRY_MOD**. The **DS_C_ENTRY_MOD** object is defined by a separate array of object descriptors whose base address is the value of the **DS_CHANGES** attribute.

There can be more than one instance of this attribute in the **DS_C_ENTRY_MOD_LIST** object. An indication of this is given by the word **LIST** in the name **DS_C_ENTRY_MOD_LIST**. Each attribute contains one separate entry modification. To learn how the modification itself is specified, see "The DS_C_ENTRY_MOD Object" on page 69. If more than one modification is specified, the modifications are performed by **ds_modify_entry()** in the order in which the **DS_CHANGES** attributes appear in the **DS_C_ENTRY_MOD_LIST** object.

The DS_C_ENTRY_MOD Object: The **DS_C_ENTRY_MOD** class object holds the information associated with a directory entry modification. (For an illustration of its structure, see Figure 18 on page 68.) Each **DS_C_ENTRY_MOD** object describes one modification. To create a list of modifications suitable to be passed to a **ds_modify_entry()** call, describe each modification in a separate **DS_C_ENTRY_MOD** object, and then insert these objects as multiple instances of the **DS_CHANGES** attribute in a **DS_C_ENTRY_MOD_LIST** object.

- **OM_CLASS**

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

- **DS_MOD_TYPE**

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

- **DS_ADD_ATTRIBUTE**

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

- **DS_ADD_VALUES**

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

- **DS_REMOVE_ATTRIBUTE**

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

- **DS_REMOVE_VALUES**

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

- **DS_ATTRIBUTE_TYPE**

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

- **DS_ATTRIBUTE_VALUES**

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

Example Definition of a DS_C_ENTRY_MOD_LIST Object: The following code fragment is an example definition of a **DS_C_ENTRY_MOD_LIST** object.

```
OM_string my_uuid;
```

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

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

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

The **DS_C_ENTRY_INFO_SELECTION** object contains the attributes shown in Figure 19.

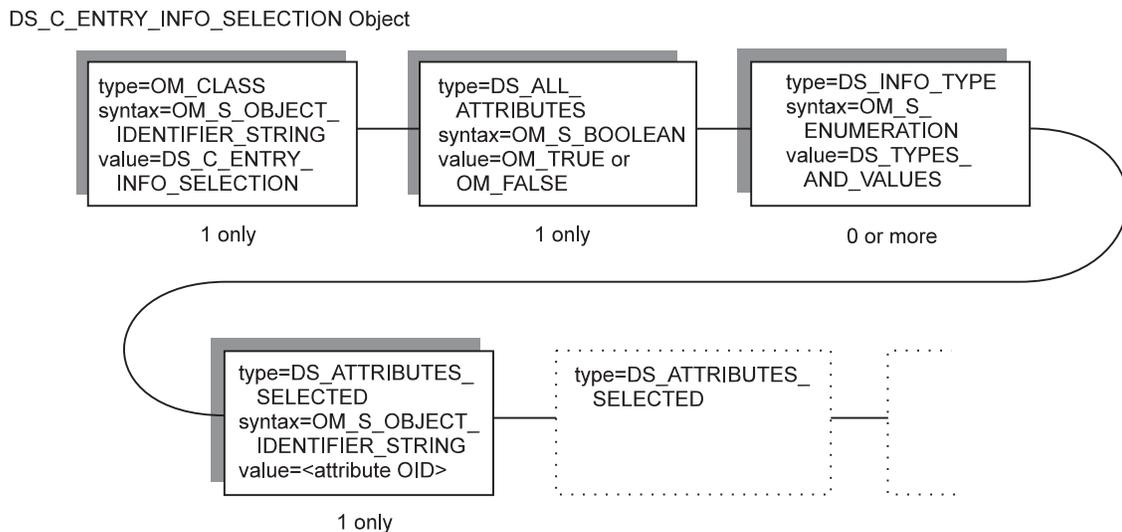


Figure 19. The DS_C_ENTRY_INFO_SELECTION Object

Note that this object class has no subobjects.

- **OM_CLASS**

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

- **DS_ALL_ATTRIBUTES**

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

- **OM_TRUE**, meaning that all attributes in the directory entry should be read. Any values specified by the **DS_ATTRIBUTES_SELECTED** attribute are ignored.
- **OM_FALSE**, meaning that only some of the entry attributes should be read; namely, those specified by the **DS_ATTRIBUTES_SELECTED** attribute.

- **DS_INFO_TYPE**

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

- **DS_TYPES_ONLY**, meaning that only the attribute types of the selected attributes should be read.
- **DS_TYPES_AND_VALUES**, meaning that both the attribute types and the attribute values of the selected attributes should be read.

- **DS_ATTRIBUTES_SELECTED**

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

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

Example Definition of a DS_C_ENTRY_INFO_SELECTION Object: The following code fragment provides an example definition of a **DS_C_ENTRY_INFO_SELECTION** object:

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

Attribute and Data Type Translation

This section provides translations between existing CDS and XDS for attributes and data types. Table 5 lists the OM syntax for CDS attributes. Table 6 lists the OM syntax for CDS data types. Table 7 on page 73 defines the mapping of the CDS Data Types to OM Syntaxes.

Table 5. CDS Attributes to OM Syntax Translation

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

Table 6 (Page 1 of 2). OM Syntax to CDS Data Types Translation

OM Syntax	CDS Data Type
OM_S_TELETEX_STRING	cds_char
OM_S_OBJECT_IDENTIFIER_STRING	cds_byte
OM_S_OCTET_STRING	cds_byte
OM_S_PRINTABLE_STRING	cds_char

Table 6 (Page 2 of 2). OM Syntax to CDS Data Types Translation

OM Syntax	CDS Data Type
OM_S_NUMERIC_STRING	cds_char
OM_S_BOOLEAN	cds_long
OM_S_INTEGER	cds_long
OM_S_UTC_TIME_STRING	cds_char
OM_S_ENCODING_STRING	cds_byte

Table 7. CDS Data Types to OM Syntax Translation

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

Part 3. GDS Application Programming

This part of the book describes the Global Directory Service (GDS) and contains conceptual materials with descriptions of programming tasks and the use of programming interfaces.

Note: z/OS DCE does not support GDS. The GDS naming information presented is intended to increase your understanding of DCE name structure and concepts.

Chapter 4. GDS API: Concepts and Overview	77	Strings	124
Directory Service Interfaces	77	Other Syntaxes	124
The X.500 Directory Information Model	78	Service Interface Data Types	124
Directory Objects	78	The OM_descriptor Data Type	125
Attribute Types	79	Data Types for XDS API Function Calls	127
Object Identifiers	80	Data Types for XOM API Calls	127
Object Entries	81	OM Function Calls	128
X.500 Naming Concepts	83	Summary of OM Function Calls	128
Distinguished Names	83	Using the OM Function Calls	129
Relative Distinguished Names and Attribute Value Assertions	84	XOM API Header Files	133
Multiple AVAs	84	XOM Type Definitions and Symbolic Constant Definitions	133
Aliases	85	XOM API Macros	133
Name Verification	86	Chapter 6. XDS Programming	137
Schemas	86	XDS Interface Management Functions	137
The GDS Standard Schema	87	The ds_initialize() Function Call	138
The Structure Rule Table	87	The ds_version() Function Call	138
The Object Class Table	89	The ds_shutdown() Function Call	140
The Attribute Table	92	Directory Connection Management Functions	140
Defining Subclasses	93	A Directory Session	140
Abstract Syntax Notation 1	93	The ds_bind() Function Call	140
ASN.1 Types	94	The ds_unbind() Function Call	141
Basic Encoding Rules	95	Automatic Connection Management	141
Chapter 5. XOM Programming	97	XDS Interface Class Definitions	141
OM Objects	97	The DS_C_CONTEXT Parameter	142
OM Object Attributes	97	Directory Class Definitions	142
Object Identifiers	100	Directory Operation Functions	143
C Naming Conventions	100	Directory Read Operations	143
Public Objects	102	Reading an Entry from the Directory	144
Private Objects	111	Step 1: Export Object Identifiers for Required Directory Classes and Attributes	144
Object Classes	111	Step 2: Declare Local Variables	145
Packages	117	Step 3: Build Public Objects	145
The Directory Service Package	117	Step 4: Create an Entry-Information-Selection Parameter	146
The Basic Directory Contents Package	118	Step 5: Perform the Read Operation	147
The Strong Authentication Package	119	Directory Search Operations	150
The Global Directory Service Package	119	Directory Modify Operations	150
The MHS Directory User Package	119	Modifying Directory Entries	151
Package Closure	120	Step 1: Export Object Identifiers for Required Directory Classes and Attributes	152
Workspaces	120	Step 2: Declare Local Variables	152
Storage Management	121		
OM Syntaxes for Attribute Values	122		
Enumerated Types	123		
Object Types	123		

Step 3: Build Public Objects	153
Step 4: Create Descriptor Lists for Attributes	154
Step 5: Perform the Operations	155
Return Codes	157

Chapter 7. Example Application

Programs	159
General Programming Guidelines	159
The example.c Program	159
The example.c Code	162
Error Handling	168
The teldir.c Program	170
Predefined Static Public Objects	170
Partially Defined Static Public Objects	171
Dynamically Defined Public Objects	172
Main Program Procedural Steps	173
The teldir.c Code	174

Chapter 8. Using Threads with the XDS/XOM API

Overview of Example Threads Program	192
User Interface	193
Input File Format	193
Program Output	193

Prerequisites	194
Description of Thradd Example Program	194
Detailed Description of Thread Specifics	195
The thradd.c Code	197
The thradd.h Header File	205

Chapter 9. XDS/XOM Convenience

Routines	209
String Handling	209
Strings Representing GDS Attribute Information	210
Strings Representing Structured GDS Attribute Information	210
Strings Representing a Structured GDS Attribute Value	212
Strings Representing a Distinguished Name	212
Strings Representing Expressions	213
Examples of strings handled by omX_string_to_object()	214
Examples of strings returned by omX_object_to_string()	216
The teldir2.c Program	217
The teldir2.c Code	218

Chapter 4. GDS API: Concepts and Overview

Note: z/OS DCE does not support GDS. The information in this chapter is presented to enable you to build the GDS portion of a DCE name. The GDS portion of a DCE name is used if the DCE cell has a GDS (X.500) cell name.

The Global Directory Service (GDS) is a distributed, replicated directory service. It is distributed because information is stored in different places in the network. Requests for information may be routed by the GDS to directory servers throughout the network. It is replicated because information can be stored in more than one location for easier and more efficient access by its users.

The GDS is based on the CCITT X.500/ISO 9594 (1988) international standard. The aim of this standard, also referred to as the OSI Directory standard, is to provide a global directory that supports network users and applications with information required for communication. The Directory plays a significant role in allowing the interconnection of information processing systems from different manufacturers, under different managements, of different levels of complexity, and of different ages.

GDS is the DCE implementation of the OSI Directory standard. Together with the Cell Directory Service (CDS) it provides its users with a centralized place to store information required for communications, which can be retrieved from anywhere in a distributed system. GDS maintains information describing objects such as people, organizations, applications, distribution lists, network hardware, and other distributed services dispersed over a large geographical area.

The CDS stores names and attributes of resources located in a DCE cell. A DCE cell consists of various combinations of DCE machines connected by a network. Each DCE cell contains its own Cell Directory Server, which provide access to local resource information. The Cell Directory Service is optimized for local information access by its users. For a more detailed description of cells and their resource services, see *z/OS DCE Introduction*.

The GDS serves as a general-purpose information repository. It provides information about resources outside a DCE cell. It ties together the various cells by helping to find remote cells. A detailed discussion of the DCE namespace, its various servers and their interaction is provided in Chapter 2, "Programming in the CDS Namespace" on page 17.

Directory Service Interfaces

XDS and XOM are application programming interfaces. XOM and XDS application interfaces are based on X/Open standards specifications. Together, these interfaces provide you with a library of functions you can use develop applications that access the Directory Service.

The XOM Application Programming Interface (XOM API) is an interface for creating, deleting, and accessing information objects. The XOM API defines an object-oriented information model. Objects belong to classes and have attributes associated with them. The XOM API also defines basic data types, such as Boolean, string, object, and so on. The representation of these objects is transparent to the programmer. Objects can be manipulated through the XOM interface.

On DCE, you use the XDS API to make Directory Service calls. In DCE, XDS API directs the calls it receives to either the GDS or the CDS by examining the names of the information objects to be looked up as shown in Figure 20 on page 78. It uses the XOM interface for defining and handling information objects. These objects are passed as parameters to the XDS routines, and from the XDS routines are received as return values. The XDS API contains functions for managing connections with a Directory Server: for reading, comparing, adding, removing, modifying, listing, and searching for directory entries.

The GDS Package provides additional information objects that provide for security and cache management when using GDS.

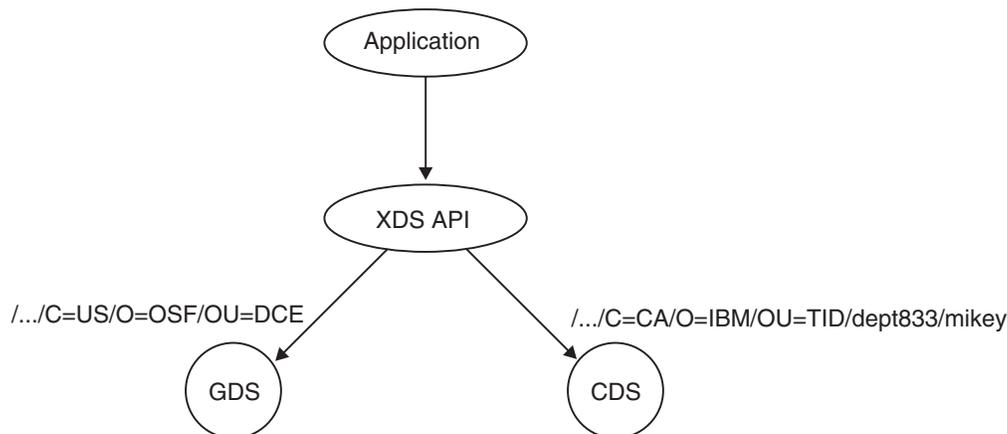


Figure 20. XDS: Interface to GDS and CDS

The X.500 Directory Information Model

This section describes the directory information model of X.500, which GDS is based on. A directory is a collection of information about some part of the world. The most familiar type of directory is the list of names and numbers that make up a city telephone directory. A name is provided with some information about the named object, such as an address and telephone number. The ISO and CCITT standards define a *directory information model* that in turn defines the abstract structure of directory information, services, and protocols for a computer network environment, such as DCE.

Directory Objects

The Directory contains information about objects. The standard defines an object very broadly as *anything in some 'world', generally the world of telecommunications and information processing or some part thereof, which is identifiable (can be named)*. Some examples of objects include people, corporations, and application processes.

Each object known to the Directory is represented by an entry. The set of all entries is called the Directory Information Base (DIB), which is a hierarchical tree. Each entry consists of a set of attributes representing specific information about the object. Each attribute, in turn, has a type and one or more values of that type. Attributes with more than one value are referred to as multi-valued or recurring attributes.

Figure 21 on page 79 shows the structure of the DIB.

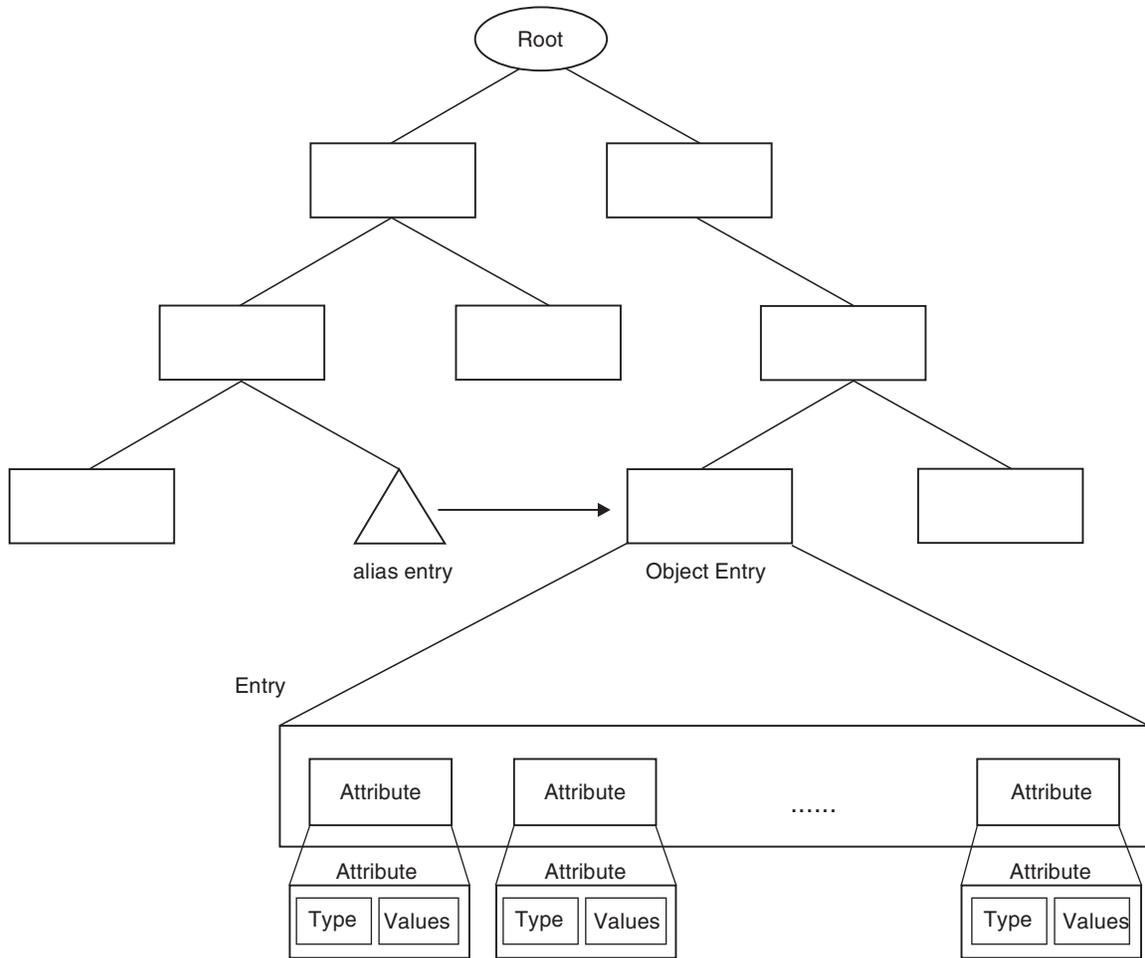


Figure 21. The Structure of the DIB

The attributes that comprise a single entry may be of various types. For example, an entry for a person may contain that person’s name, address, and phone number. If the person has a second telephone number, the attribute of type telephone number may have two values, one for each telephone number.

Object entries are composed of mandatory and optional attributes. Mandatory and optional attributes are discussed in “The Object Class Table” on page 89.

Attribute Types

All attributes in a particular entry must be of different attribute types. Each attribute type is assigned a unique object identifier value. The Directory standard assigns object identifiers for several commonly used attribute types, including surname, country name, telephone number, and presentation address. Other international standards may define additional attribute types. For example, the X.400 Message Handling standard defines mail specific attributes like O/R address. It is expected that various national and private organizations will also define attribute types of their own. The CDS attributes (defined in the **xdscds.h** header file) and the GDS Package attributes (defined in the **xdsgds.h** header file) are examples of additional attribute definitions.

Object Identifiers

Objects in a network environment, such as DCE, require unique names to distinguish them from one another. To provide these names, object identifiers are allocated by an administrative organization, such as a standards body. An object identifier is a hierarchical sequence of numbers uniquely identifying an object. Associated with each object identifier is a character string to make it easier to document.

The possible values of object identifiers are defined in a tree. Part of this tree is shown in Figure 22. It begins with three numbered branches coming from the root: branch 0, assigned to CCITT; branch 1, assigned to ISO; and branch 2, a joint ISO-CCITT branch. Below each of these branches are other numbered branches assigned to various standards such as the Directory Service (**ds(5)**) and Electronic Mail Service (**mhs-motis(6)**) with each ending in a named object. Thus, the name of any of these objects is a series of integers describing a path down this tree to the leaf node.

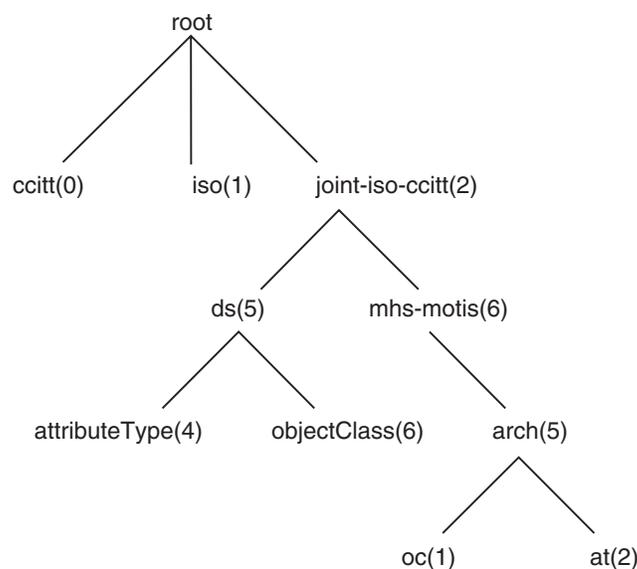


Figure 22. Object Identifiers

The object identifier associated with the XDS Directory Service Package is defined as follows:

```
{iso(1) identified-organization(3) icd-ecma(12) member-company(2) dec(1011) xopen(28) dsp(0)}.
```

All object classes and object attributes in the Directory Service Package have these numbered branches associated with them. The classes and attributes, in turn, have their own unique numbers. These object identifiers are defined in header files included as part of the XDS and XOM API software. For example, the attribute type **Common Name** is identified by the object identifier 2.5.4.3.

Table 8 contains a sample list of object identifiers for selected attributes. The complete list is provided in Chapter 12, “Basic Directory Contents Package” on page 275.

Table 8 (Page 1 of 2). Object Identifiers for Selected Attribute Types

Attribute Type	Object Identifier
Aliased Object Name	2.5.4.1
Business Category	2.5.4.15
Common Name	2.5.4.3

Table 8 (Page 2 of 2). Object Identifiers for Selected Attribute Types

Attribute Type	Object Identifier
Country Name	2.5.4.6
Description	2.5.4.13

Note: The object identifiers in Table 8 on page 80 stem from the root **{joint-iso-ccitt(2) ds(5) attributeType(4)}**.

Object Entries

Entries are grouped into generic object classes based on the type of object they represent. Examples of object classes are Country, Organization Person, and Application Entity. All entries contain a special attribute, called the object class attribute, indicating to which object class (or classes) they belong.

Entries that model a certain object and contain information about the object in terms of attributes are called **object entries**. The Directory contains a second type of entry which is a pointer to an object entry called an **alias entry**. Alias entries are discussed in “Aliases” on page 85.

In summary (as shown in Figure 21 on page 79), the DIB is made up of entries, each of which contains information about objects. Entries consist of attributes; each attribute has a type and one or more values.

“X.500 Naming Concepts” on page 83 describes how objects are organized in the DIB using the Directory Information Tree (DIT). Figure 23 on page 82 shows an example of an entry describing Organizational Person.

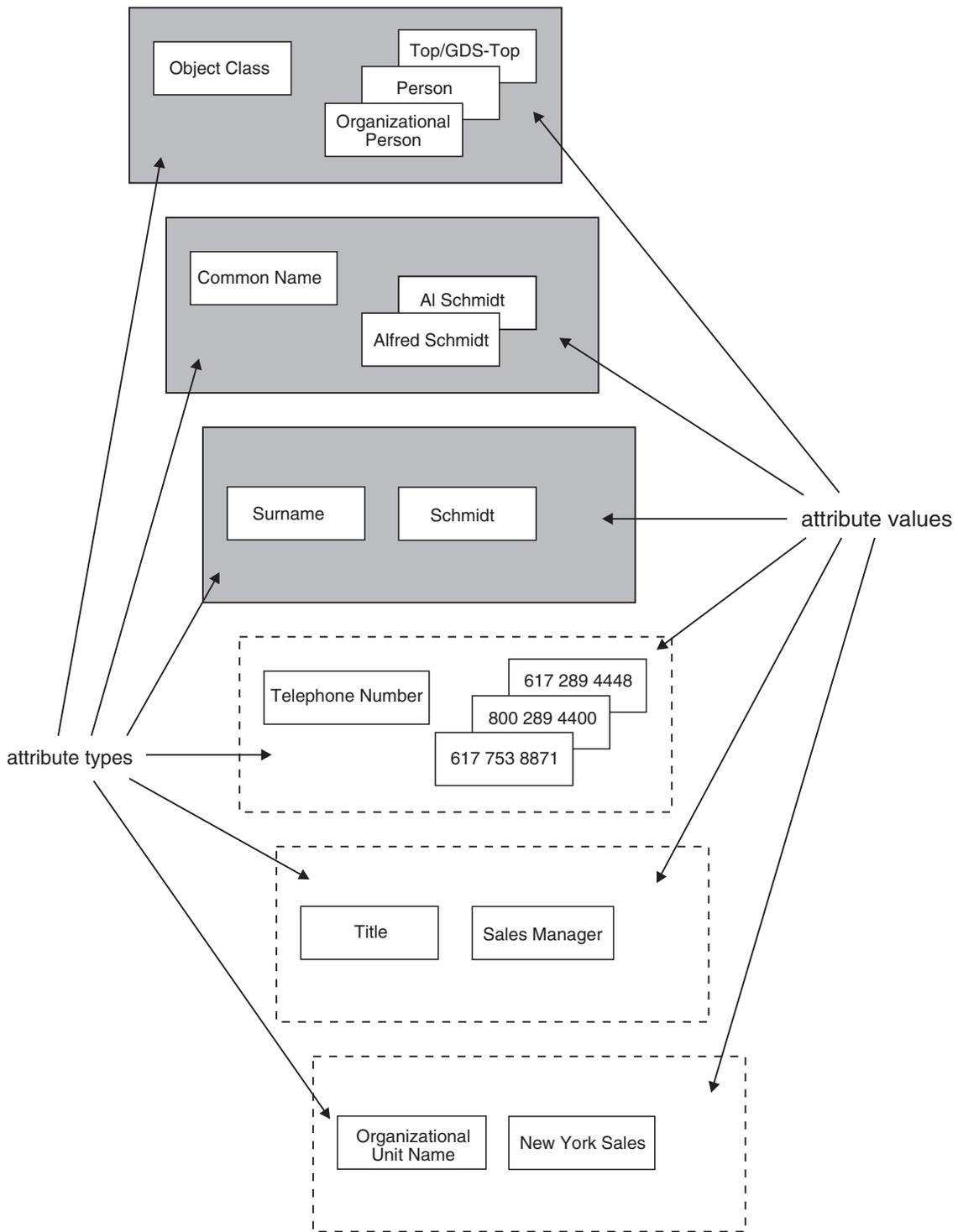


Figure 23. A Directory Entry Describing Organizational Person

X.500 Naming Concepts

Large amounts of information need to be organized in some way to make efficient retrieval possible and ensure that names are unique. Information in the DIB is organized into a hierarchical structure known as the Directory Information Tree (DIT). The structure and naming of the nodes in the DIT are specified by registration authorities for a standardized set of X.500 names and by implementers of the directory service (such as OSF) for implementation-specific names. The DIT hierarchy is described by a schema. Schemas are described in more detail in “Schemas” on page 86.

Although the X.500 standard does not mandate a specific schema, it does make general recommendations. For example, countries and organizations should be named close to the root of the DIT; people, applications, and devices should be named further down in the hierarchy.

Distinguished Names

A hierarchical path exists from the root of the DIT to any entry in the DIB. To access information stored in an entry, a name that uniquely describes that entry must be given. An RDN distinguishes an entry from other entries with the same superior node in the DIT. A sequence of RDNs, starting from the root of the tree, can identify a unique path down the tree, and thus a unique entry. This sequence of RDNs, each of which identifies a particular entry, is the distinguished name of that entry. Each entry in the DIB can be referenced by giving its distinguished name.

Figure 24 shows an example of a distinguished name. The shaded boxes in the DIT represent the entries that are named in the column labeled RDN (Relative Distinguished Name). The schema dictates that countries are named directly below the root, followed by organizations, organization units, and people.

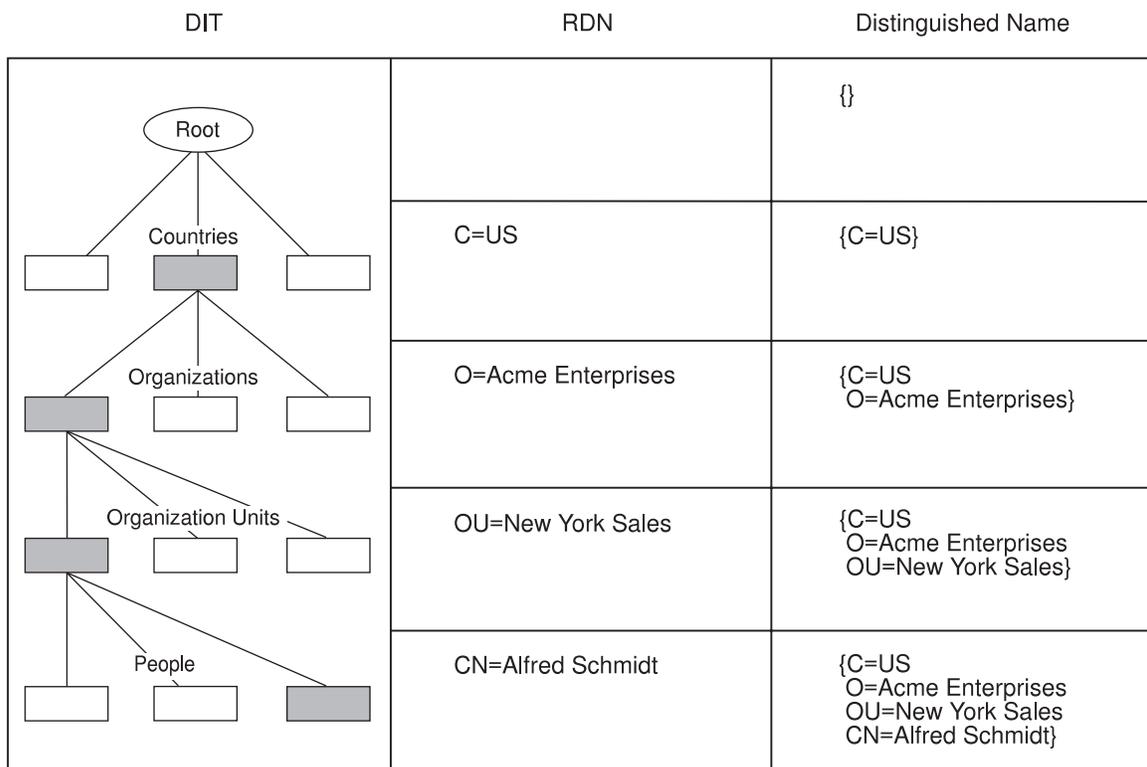


Figure 24. A Distinguished Name in a Directory Information Tree

Every entry in the DIB has a distinguished name, not just the leaf nodes. For example, the entry for the Organization, Acme Enterprises is represented by the shaded box in the Organizations subtree. Its distinguished name is the concatenation of the distinguished name of the entry above with its relative distinguished name. The entry for People, Alfred Schmidt, is represented by the shaded box in the People subtree.

Relative Distinguished Names and Attribute Value Assertions

Each entry has a unique **Relative Distinguished Name** (RDN), which distinguishes it from all other entries with a particular immediate superior in the DIT.

An RDN consists of one or more assertions of the type and value of an attribute. A pair consisting of an attribute type and a value of that type is known as an Attribute Value Assertion (AVA). All attribute types in an RDN must be different. The attribute value of an attribute in an RDN's AVA is called the distinguished value of that attribute, as opposed to the other possible values of that attribute.

The assertion is TRUE if the entry contains an attribute of the specified type, and if one of that attribute's values matches the AVA's distinguished attribute value. An entry commonly has a RDN which will consist of a single AVA. In some cases, however, more than one AVA may be required to distinguish an entry. (Multiple AVAs are discussed in "Multiple AVAs.")

The entry shown in Figure 23 on page 82 contains the RDN: **Common Name = Alfred Schmidt**. The attribute consists of three values: Alfred Schmidt, A. S. Schmidt, and Al Schmidt. The AVA **Common Name = Alfred Schmidt** contains the value Alfred Schmidt, which has been designated as the distinguished value in the AVA.

Multiple AVAs

Frequently, as shown in the previous section, an entry contains a single distinguished value; the RDN therefore comprises a single AVA. However, under certain circumstances, additional values (and hence multiple AVAs) may be used.

Figure 23 on page 82 shows the contents of an entry describing Organizational Person. The RDN of an Organizational Person entry is usually composed of a single AVA such as the Common Name attribute type with a distinguished value (in Figure 24 on page 83, the **AVA CN = Alfred Schmidt**). Depending on the schema, the RDN of an Organizational Person entry may contain more than one AVA. For example, the RDN in Figure 24 on page 83 could contain the AVAs **CN = Alfred Schmidt, OU = New York Sales** with Alfred Schmidt and New York Sales as distinguished values.

In summary:

- A DIT consists of a collection of distinguished names.
- Distinguished names result from a concatenation of the RDNs.
- RDNs consist of an unordered collection of attribute type and value pairs (AVAs).

Aliases

An alternative name or alias is supported in the DIT by the use of special pointer entries called alias entries. Alias entries do not contain any other attributes beyond their distinguished attributes, the object class attribute, and the aliased object name attribute, that is, the distinguished name of the aliased object entry. Furthermore, an alias entry has no subordinate entries, making it, by definition, a leaf entry of the DIT as shown in Figure 25. Alias entries point to object entries and provide the basis for alternative names for the corresponding objects.

Aliases are used to do such things as provide user-friendly names, direct the search for a particular entry, reduce the scope of a search, provide for common alternative abbreviations and spellings, or provide continuity after a name change.

Figure 25 demonstrates how an alias name provides continuity after a name change. The ABC company's branch office located originally in Osaka has moved to Tokyo. To make the transition easier for Directory Service users and to guarantee that a search based on the old information will find its target, an alias for **O=ABC** has been added to the directory beneath **L=Osaka**. This alias entry points to the object entry **O=ABC**. A search for ABC under **L=Osaka** in the DIT finds the entry: **/C=Japan/L=Tokyo/O=ABC**.

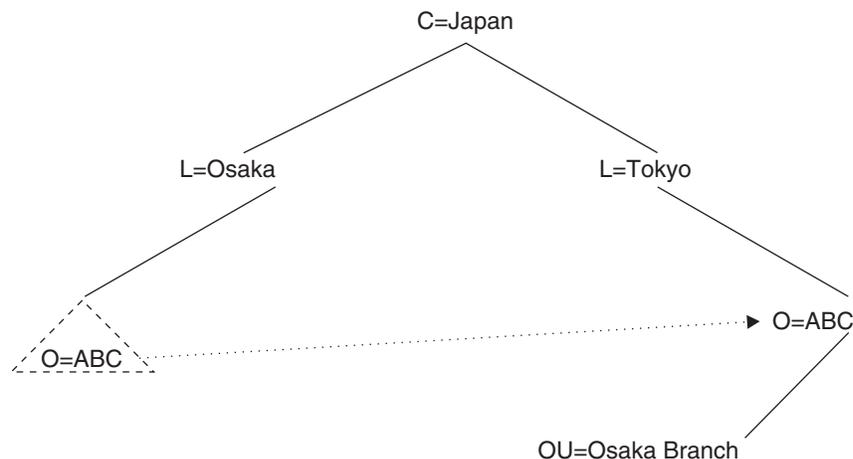


Figure 25. An Alias in the Directory Information Tree

Another use of alias entries is as an alternative to **filtering**, that is, using assertions about particular attributes to search through the DIT. Although this approach does not require any special information to be set up in the DIT, it may be expensive to search where there is a large population of entries and attributes. An alternative approach is to set up special subtrees whose naming structures are designed for *Yellow Pages* type searching. Figure 26 on page 86 shows an example of such a subtree populated by alias entries only. In reality, the entries within these subtrees may be a mixture of object and alias entries, so long as there exists only one object entry for each object stored in the directory.

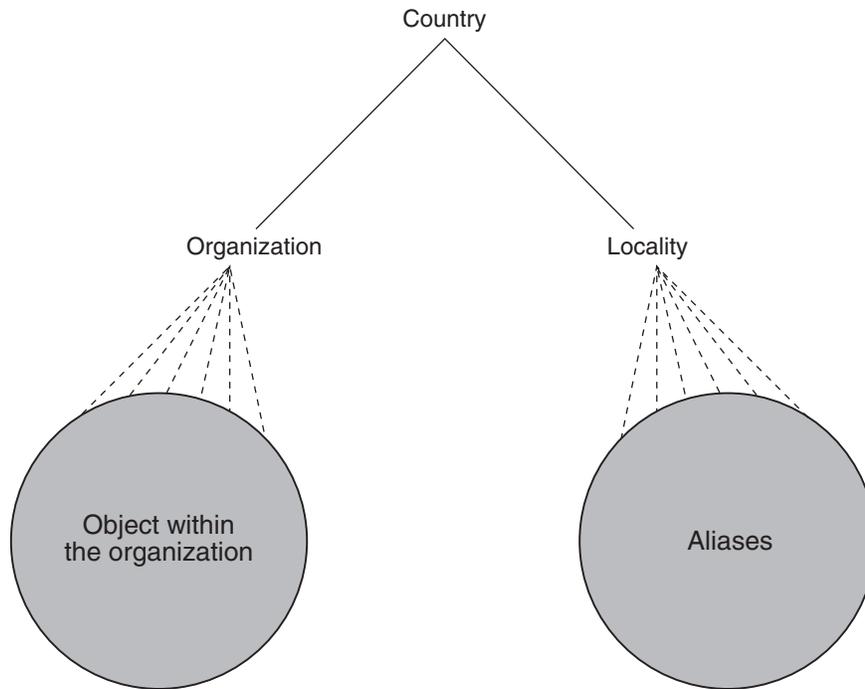


Figure 26. A Subtree Populated by Aliases

An object with an entry in the DIT can have zero or more aliases. Several alias entries can point to the same object entry. An alias entry can point to an object that is not a leaf entry. Only object entries can have aliases. Thus, aliases of aliases are not permitted.

Name Verification

A Directory user identifies an entry by supplying an ordered set of RDNs (each of which consists of an unordered set of AVAs) forming a purported name. The purported name is mapped onto the desired entry by the process of name verification, which performs a distributed tree walk through the DIT. When a purported name is a valid name, a distinguished name exists with the same number of RDNs and matching AVAs within the RDNs.

Schemas

The structure of directory information is governed by a set of rules called a **schema**. Schemas specify rules for the following:

- The structure of the DIT
- The contents of entries in terms of attributes
- The syntax of attribute values and rules for comparing and matching them

The GDS Standard Schema

The DCE software package includes a default or *standard* schema for GDS. This is the GDS proprietary interpretation of the X.500 schema.

Each attribute in the schema is assigned a unique object identifier and the syntax of its value. In addition, the schema specifies the mechanism by which attributes of this type are compared with one another. Each entry in the DIT belongs to an object class governed by the schema. Object class definitions can be used to derive subclasses, supporting the inheritance and refinement of the attribute types defined for the super class.

Included with the GDS standard schema are the following tables that define the structure of the Directory.

- Structure Rule Table (SRT)
- Object Class Table (OCT)
- Attribute Table (AT)

The Structure Rule Table

The Structure Rule Table (SRT) specifies the relationship of object classes in the structure of the Directory. The SRT supplied with the GDS standard schema contains the entries shown in Table 9.

Table 9. Structure Rule Table Entries

Rule Number	Superior Rule Number	Acronyms of Naming Attribute	Object Class
1	0	CN	Schema
2	0	C	Country
3	2	O	Organization
4	3	OU	Organizational Unit
5	4	CN	Organizational Person
6	4	CN,OU	Organizational Person
7	4	CN	Organizational Role
8	4	CN	MHS Distribution List
9	4	CN	Application Process
10	9	CN	Application Entity
11	9	CN	DSA
12	9	CN	MHS Message Store
13	9	CN	MHS Message Transfer Agent
14	9	CN	MHS User Agent
15	2	L	Locality
16	15	CN	Residential Person
17	15	CN,STA	Residential Person

The SRT determines how the object classes are laid out in the DIT by assigning rule numbers to each object class. The Superior Rule Number of an object class specifies the object class directly above it in the DIT.

For example, the object class Organization (ORG) has a Superior Rule Number of 2, indicating that it is located in the DIT beneath the object class Country (C), which has a Rule Number of 2. Organization Unit (OU) is located beneath Organization because it has a Superior Rule Number of 3 and so forth.

The SRT only contains structured object classes, that is, classes forming branches in the DIT. Other object classes, such as abstract and alias classes, are not included.

The SRT specifies the attributes used to name entries belonging to each object class. These attributes, called **nam^{ing} attributes**, are used to define the RDN and therefore the distinguished name of directory entries.

Figure 27 shows the structure of the DIT as defined by the SRT of the GDS standard schema.

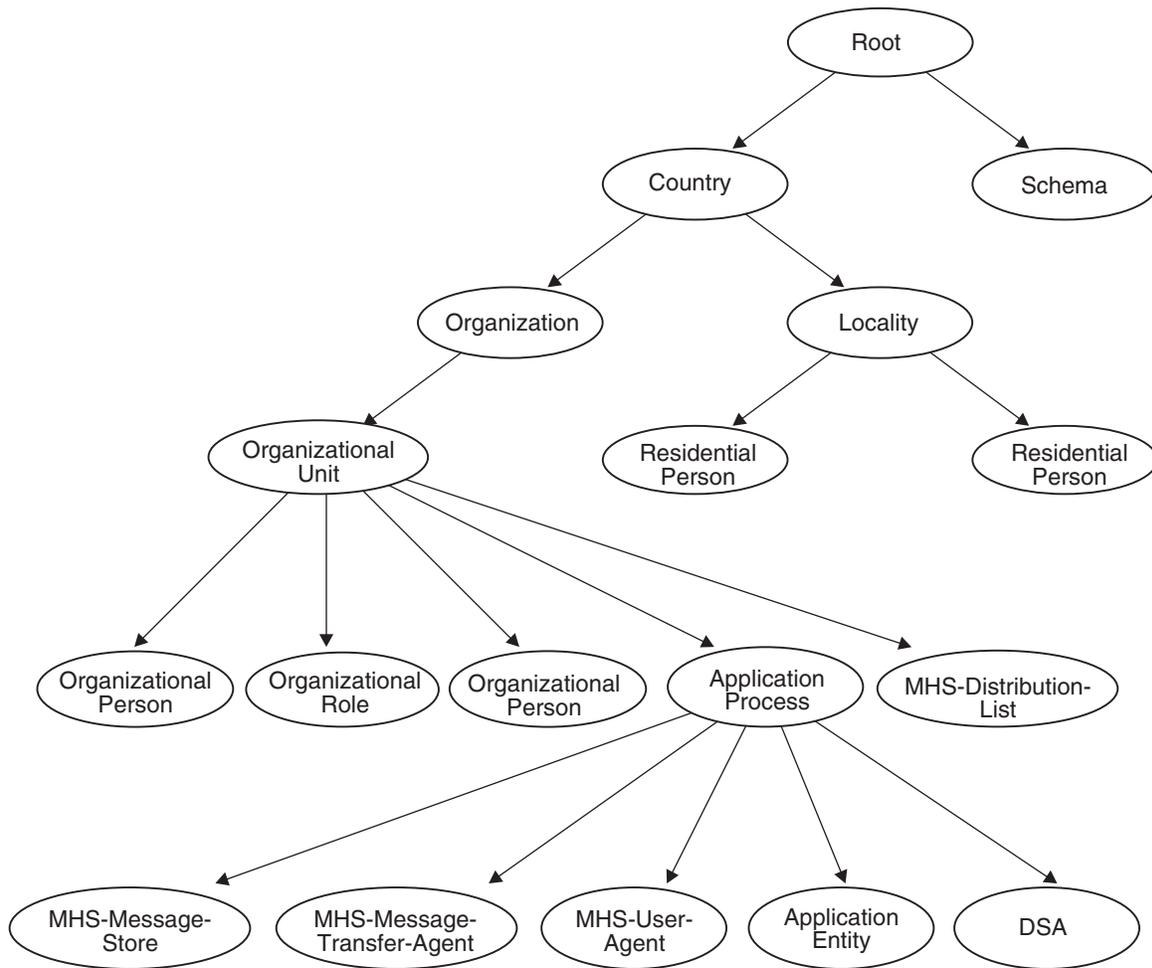


Figure 27. SRT DIT Structure for the GDS Standard Schema

The Object Class Table

The object classes that make up the GDS standard schema are defined in the OCT. Table 10 contains a partial listing of the OCT. Refer to the *z/OS DCE Administration Guide* for a complete listing of the OCT for the GDS standard schema. Each column in Table 10 contains information about an object class entry in the schema.

Table 10. Object Class Table Entries

Object Class Acronym	Super Class Acronyms	Object ID	Object Class Name	Object Class Kind	File No.	Auxiliary Object Classes	Mandatory Attributes	Optional Attributes
TOP	None	85.6.0	Top	Abstract	-1	-	OCL	None
ALI	TOP	85.6.1	Alias	Alias	-1	-	AON	None
C	GTP	85.6.2	Country	Structural	1	-	C	DSC SG CDC CDR
LOC	GTP	85.6.3	Locality	Structural	4	-	None	DSC L SPN STA SEA SG CDC CDR
ORG	GTP	85.6.4	Organization	Structural	1	-	O	DSC L SPN STA PDO PA PC POB FTN IIN TN TTI TXN X1A PDM DI RA SEA UP BC SG CDC CDR

Note: All these object identifiers stem from the root **{joint-iso-ccitt(2) ds(5) objectClass(6)}**. The **joint-iso-ccitt(2)** and the **ds(5)** are combined to form the number **85** as shown in Table 10. In general, first two decimal numbers of an OID are combined via the equation $40 * A + B$ to yield a single number representing the first two numbers. In this case, $40*2+5=85$.

Column 2, **Superclass Acronyms**, provides the class from which an object class inherits its attributes. Using the information in Column 2, it is possible to derive a graphical representation of the inheritance properties of object classes in the DIT as shown in Figure 28 on page 90.

The object class, Top, is the root of the tree, with Alias and GDS-Top as the main branches. Top contains the attribute type object class, which is inherited by all the other object classes.

Do not confuse the information in the OCT with that presented in the SRT. There is no direct relationship between the relative location of branches and leaves in the DIT structure and the inheritance properties of classes with their superclasses and subclasses.

For example, when a Directory Service request is made by a directory user, such as a read operation, the SRT is used by the Directory Service to indicate its position in the DIT. The Directory Service uses the information defined in the SRT for tree traversal so that the requested object can be located in the Directory. Figure 27 on page 88 shows the object class Organization located beneath Country in the DIT.

On the other hand, the OCT defines, among other things, the attributes of an object class along with its inherited attributes from its superclass. The superclass, in turn, inherits the attributes from its superclass, and so on until the root, Top, is reached (from which all classes derive their attributes). Figure 28 on page 90 shows the object class Organization as a subclass of GDS-Top. As such, it inherits its attributes from GDS-TOP, which in turn inherits from its superclass, Top.

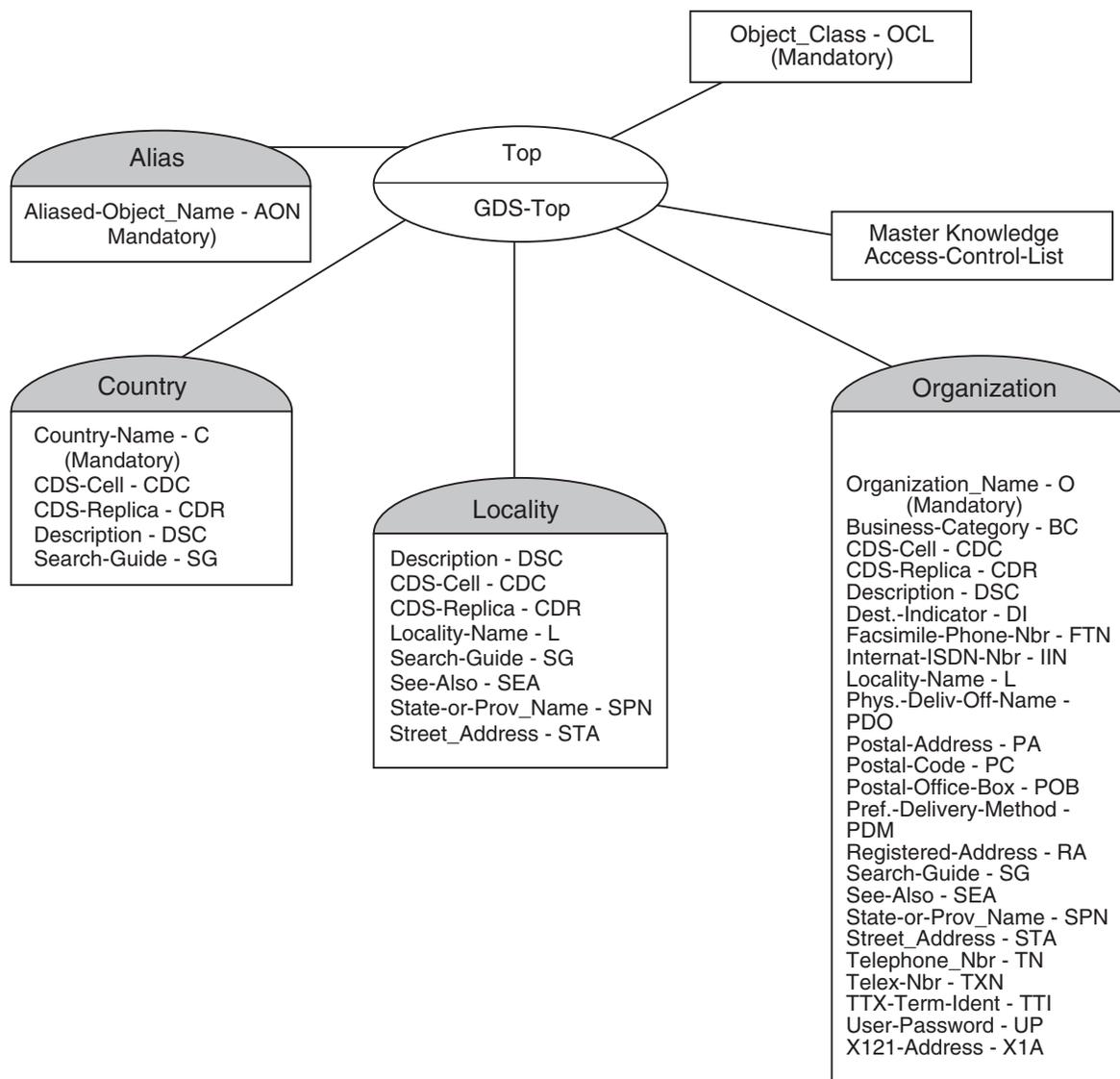


Figure 28. A Partial Representation of the Object Class Table

The OCT also contains the unique object identifier of each class in the DIT. These numbers are defined by various standards authorities and in the X.500 standards documents mentioned previously. The Attribute Table (AT) contains the predefined object identifiers for each attribute in the Directory. These object identifiers are defined in the header files that are included as part of the GDS API. Table 11 shows some examples of object identifiers for directory classes as defined in the X.500 standard.

Table 11 (Page 1 of 2). Object Identifiers for Selected Classes

Object Class Type	Object Identifier
Alias	85.6.1
Application Entity	85.6.12
Application Process	85.6.11
Country	85.6.2
Device	85.6.14

Table 11 (Page 2 of 2). Object Identifiers for Selected Classes

Object Class Type	Object Identifier
DSA	85.6.13
Group of Names	85.6.9
Locality	85.6.3
Organization	85.6.4
Organizational Person	85.6.7
Organizational Role	85.6.8
Organizational Unit	85.6.5
Person	85.6.6
Residential Person	85.6.10
Top	85.6.0

Note: All these object identifiers stem from the root **{joint-iso-ccitt(2) ds(5) objectClass(6)}**.

For each object class, there are four possible kinds: Abstract, Structural, Auxiliary, and Alias. Abstract objects are used to provide inheritance, and do not exist. Structural objects, however, do exist, and are found in the DIT and represent real-world objects. Auxiliary object classes are used to add attributes to an entry of a standardized class. For instance, message handling systems use the auxiliary class MHS User to add message handling attributes to the Organizational Person object. Alias object classes are used to create entries that point to other entries.

The file number for an object determines the file in the Directory Information Base (DIB) that will store the object. Abstract objects will have a value of -1 because they do not exist. The schema object is stored in file 0, and all other objects are stored in files with a higher file ID.

The Auxiliary Object Classes column is used to specify the Auxiliary Object classes that are associated with an object class.

Another important feature of the OCT is the distinction made between mandatory and optional attributes for each object class. This distinction is based on recommendations from X.500 standards documents. These documents (Recommendations X.520 and X.521) define selected object classes and associated attribute types using ASN.1 notation. Most object classes have one or more mandatory attributes associated with them for use by implementers who want to comply with the X.500 standards recommendations. In addition, optional attributes are defined.

The following example provides a flavor of ASN.1 notation; it shows how the object class **country** is described in Recommendation X.520 (*The Directory: Selected Object Classes*):

```
country OBJECT-CLASS
  SUBCLASS of top
  MUST CONTAIN {
    countryName}
  MAY CONTAIN {
    description,
    searchGuide}
  ::= {objectClass 2}
```

This ASN.1 definition defines **country** as a subclass of superclass **top**. The class, **country**, must contain the mandatory attribute **countryName** (or **country-name** as defined in the GDS standard schema), and

may contain the optional attributes, **description** and **searchGuide**. In addition, the DCE implementation adds two more attributes, **CDS-Cell** and **CDS-Replica**, to incorporate other aspects of the DCE environment that are implementation specific.

Country is assigned the object identifier 2.5.6.2. This number distinguishes it from the other object classes defined by the standard. The Top superclass is designated as 2.5.6.0. The first three numbers, 2.5.6, identify the object class as a member of a discrete set of object classes defined by X.500. The last number in the object identifier distinguishes objects within that discrete set. Alias, a subclass of Top, is assigned the number 2.5.6.1. Country is assigned the number 2.5.6.2, and so on. GDS-Top has no object identifier because it is implementation specific and thus not identified by the standard.

The Attribute Table

The attributes that make up the entries in the GDS standard schema are defined in the Attribute Table (AT). (Refer to the *z/OS DCE Administration Guide* for a complete listing of the AT.) The object identifiers are in the range from 85.4.0 through 85.4.35 as defined by the X.500 standard, 86.5.2.0 through 86.5.2.10 as defined by the X.400 standard, and there are additional object identifiers for GDS specific attributes.

Table 12 shows a partial listing of the attribute table for the GDS standard schema.

Note: The access class for every attribute listed in Table 12 is 0 (zero).

Table 12. Attribute Table Entries

Attribute Acronym	Object ID	Attribute Name	Lower Bound	Upper Bound	No. of Recurr. Values	Attr. Syntax	Phon. Match.	Access Class	Index Level
OCL	85.4.0	Object-Class	1	28	0	2	0	0	0
AON	85.4.1	Aliased-Object-Name	1	1024	1	1	0	0	0
KNI	85.4.2	Knowledge-Information	1	1024	0	4	0	0	0
CN	85.4.3	Common-Name	1	64	2	4	1	0	1
SN	85.4.4	Surname	1	64	2	4	1	0	0
SER	85.4.5	Serial-Number	1	64	2	5	0	0	0
C	85.4.6	Country-Name	2	2	1	1010	1	0	1
L	85.4.7	Locality-Name	1	128	2	4	1	0	1
SPN	85.4.8	State-or-Province-Name	1	128	2	4	1	0	0

The columns with the headings Lower Bound and Upper Bound specify the range of the number of bytes that the value of an attribute can contain. The schema puts constraints on the number of values that an attribute can contain in the Number of Recurring Values column.

The Attribute Syntax column describes how the data is represented and relates to ASN.1 syntax definitions for attributes. For example, a sample of ASN.1 notation for the Common-Name attribute:

```
commonName ATTRIBUTE
  WITH ATTRIBUTE-SYNTAX
    caseignoreStringSyntax
      (SIZE(1..ub-common-name))
  ::= (attributeType 3)
```

The **commonName** attribute is defined as case insensitive. The size of the string is from 1 to the upper bound defined by the schema for the **commonName** attribute in the Upper Bound column (in this case, 64 bytes).

Note also that the **commonName** attribute is assigned the number 3 by the standard. This corresponds to the 3 in the object identifier 85.4.3.

The other columns in the AT refer to the phonetic matching flag, security access classes, and index level. Security access classes refer to the three classes of attributes in the Directory Information Tree that have been defined, for which access protection is available. These classes are **Public**, **Standard** and **Sensitive**. An index level is used to determine the priority of an attribute in search queries of an object.

As mentioned for object classes, object identifier values specified in the AT are defined as constants in the GDS header files.

Defining Subclasses

The ability to define subclasses is a powerful feature of the Directory. Structure rules govern which object classes can be children of which others in the DIT and therefore determine possible name forms.

The directory standard defines a number of standard attribute types and object classes. For example, the attribute types Common Name and Description, and the object classes Country and Organization Person are defined. Implementations of the directory standard, such as DCE, define their own schemas following rules stated in the standard with additional attribute types and object classes.

Figure 29 shows the relationship between schemas and the directory information model.

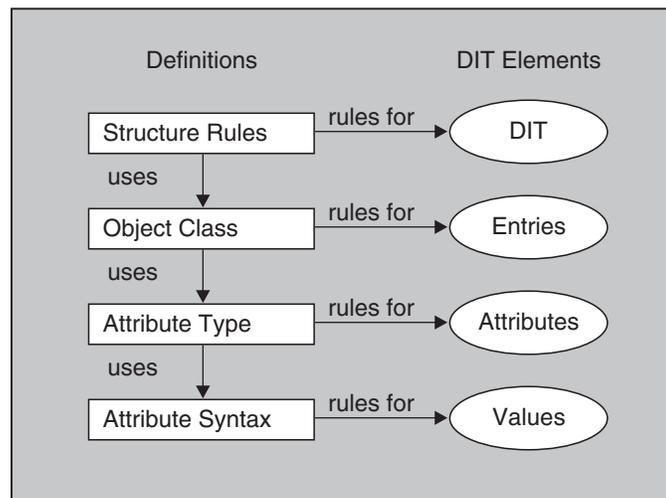


Figure 29. The Relationship Between Schemas and the DIT

Abstract Syntax Notation 1

The need for Abstract Syntax Notation 1 (ASN.1) arises because different computer systems represent information in different ways. For example, one computer may use EBCDIC character representation while another may use ASCII. To transfer a file of characters from one system to another, common representation must be used during the transfer. This transfer can be one representation or some mutually agreed upon representation negotiated by the two systems. Similarly, floating-point values, integers, and other types of data may be stored internally in different ways. To exchange information, a common format must be agreed to before information can be exchanged.

There is also a need for mapping between the many diverse representations that may exist within a network environment. To address this need, the ISO standards committee defined Abstract Syntax Notation 1 (ASN.1) (documented in ISO 8824) and Basic Encoding Rules (BER) (documented in ISO 8825).

ASN.1 is based on the idea that the aspects of transferred information that are preserved are type, length, and value. Data types are collections of values distinguished for some reason, such as characters, integers, and floating point values. Records and structure types become more complex when they combine several types into a single structure.

ASN.1 provides a way to group types into abstract syntaxes. An abstract syntax is a named group of types. The standard defines abstract syntax as the notation rules that are independent of the encoding technique used to represent them. Abstract syntax does not specify how to represent values of types, but merely defines the types that make up the group of types.

Abstract syntaxes are not enough to define how values of the data types in a specific abstract syntax are to be represented during communications. For this reason, ISO further defines a transfer syntax for each abstract syntax. A transfer syntax is a set of rules for encoding values of some specified group of types. These rules form the so-called basic encoding rules.

ASN.1 Types

ASN.1 is similar to a high-level programming language. Unlike other high-level languages, ASN.1 has no executable statements. It includes only language constructs required to define types and values. (This language is completely specified in ISO 8824.)

ASN.1 defines a number of built-in types. Users of ASN.1 can then define their own types based on the built-in types provided by the language. The ASN.1 standard defines four categories of types that are commonly used in defining application interfaces such as XOM and XDS:

- ASN.1 Simple Types
- ASN.1 Useful Types
- ASN.1 Character String Types
- ASN.1 Type Constructors

ASN.1 simple types are Bit String, Boolean, Integer, Null, Object Identifier, Octet String, and Real. Table 13 shows the relationship of OM syntaxes (syntaxes defined in XOM API) to ASN.1 simple types. (Refer to Chapter 17, “Information Syntaxes” on page 325 for the complete set of tables for the four categories of ASN.1 types.) As shown in Table 13, for every ASN.1 type except Real, there is an OM syntax that is functionally equivalent to it. The simple types are listed in the first column of the table; the corresponding syntaxes are listed in the second column.

ASN.1 Type	OM Syntax
Bit String	String(BIT_STRING)
Boolean	BOOLEAN
Integer	INTEGER
Null	NULL
Object Identifier	String(OBJECT_IDENTIFIER_STRING)
Octet String	String(OCTET_STRING)

Table 13 (Page 2 of 2). Syntax for the Simple ASN.1 Types

ASN.1 Type	OM Syntax
Real	None

An example illustrates how OM syntaxes are used to define the syntax of values for various attributes.

One of the simplest of the ASN.1 types is Boolean. There are only two possible values for a Boolean type: TRUE and FALSE. The **DS_FROM_ENTRY** OM attribute of the **DS_C_ENTRY_INFO** object class has a value syntax of **OM_S_BOOLEAN**. **OM_S_BOOLEAN** is the C language representation for the OM syntax that corresponds to the ASN.1 Boolean type. The value of the **DS_FROM_ENTRY** OM attribute indicates whether information from the Directory was extracted from the specified object's entry (TRUE), or from a copy of the entry (FALSE). The actual C language definition for **OM_S_BOOLEAN** is made in the XOM API header file **xom.h**.

Basic Encoding Rules

It is possible to define a single transfer syntax that is powerful enough to encode values drawn from a number of abstract syntaxes. ISO defines a set of rules for encoding values of many different types for ASN.1. This set of encoding rules is called Basic Encoding Rules (BER). It is so powerful that values from any abstract syntax described using ASN.1 can be encoded using the transfer syntax defined by BER.

Although other transfer syntaxes could be used for representing values from ASN.1, BER is used most often. The transfer syntax is completely described in ISO 8825.

Chapter 5. XOM Programming

XOM API defines a general-purpose interface for use in conjunction with other application-specific APIs for OSI services, such as XDS API to Directory Services or X.400 Application API to electronic mail service. It presents the application programmer with a uniform information architecture based on the concept of groups, classes, and similar information objects.

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

Complete XDS example programs that you can find in Chapter 7, “Example Application Programs” on page 159 are:

- **example.c (example.h)**
- **teldir.c**

OM Objects

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

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

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

OM Object Attributes

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

An OM attribute type is a category into which all the values of an OM attribute are placed on the basis of their purpose. Some OM attributes may either have zero, one, or multiple values. The OM attribute type is used as the name of the OM attribute. An OM attribute value is an information item that can be viewed as a characteristic or property of the OM object of which it is a part.

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

OM Attribute types and syntaxes have integer values and symbolic equivalents assigned to them by naming authorities in the various API specifications. The integers that are assigned to the OM attribute

type and syntax are fixed, but the attribute value may change. These OM attribute types and syntaxes are defined in the DCE implementation of XDS and XOM APIs in header files that are included with the software along with additional OM attributes specific to the GDS implementation.

For example, the tables in Figure 30 on page 99 show the OM attributes, syntax, and values for the OM class, **DS_C_ENTRY_INFO_SELECTION**, and how the integer values are mapped to corresponding names in the **xom.h** and **xds.h** header files. The GDS portion of Part 1, “Using the DCE Directory APIs” contains a table for every OM class supported by the Directory Service. Refer to Chapter 11, “XDS Class Definitions” on page 241 for a complete description of **DS_C_ENTRY_INFO_SELECTION** and the accompanying table.

DS_C_ENTRY_INFO_SELECTION is a subclass of **OM_C_OBJECT**. This information is supplied in the description of this OM class in Chapter 11, “XDS Class Definitions” on page 241. As such, **DS_C_ENTRY_INFO_SELECTION** inherits the OM attributes of **OM_C_OBJECT**. The only OM attribute of **OM_C_OBJECT** is **OM_CLASS**. It identifies the object’s OM class, which, in this case, is **DS_C_ENTRY_INFO_SELECTION**. **DS_C_ENTRY_INFO_SELECTION** identifies information to be extracted from a directory entry and has the following OM attributes, in addition to those inherited from **OM_C_OBJECT**:

- **DS_ALL_ATTRIBUTES**
- **DS_ATTRIBUTES_SELECTED**
- **DS_INFO_TYPE**.

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

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

OM_S_OBJECT_IDENTIFIER_STRING contains an octet string of integers that are BER encoded object identifiers of the types of OM attributes in the OM attribute list. The value of **DS_ATTRIBUTES_SELECTED** is only significant if the value of **DS_ALL_ATTRIBUTES** is **OM_FALSE**.

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

OM Attributes of a DS_C_DS_DN

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

OM Attributes of a DS_C_ENTRY_INFO_SELECTION

Attribute	Value Syntax	Value Length	Value No.	Value Initially
DS_ALL_ATTRIBUTES	OM_S_BOOLEAN	-	1	OM_TRUE
DS_ATTRIBUTES_SELECTED	String(OM_S_OBJECT_IDENTIFIER_STRING)	-	0 or more	-
DS_INFO_TYPE	Enum(DS_Information_Type)	-	1	DS_TYPES_AND_VALUES

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

sample code from
the xom.h header file

```
enum DS_Information_Type {
    DS_TYPES_ONLY = 0,
    DS_TYPES_AND_VALUES = 1
};
```

-
-
-

```
#define DS_ALL_ATTRIBUTES ((OM_TYPE) 707)
#define DS_ATTRIBUTES_SELECTED ((OM_TYPE) 710)
#define DS_INFO_TYPE ((OM_type) 734)
```

sample code from
the xds.h header file

Figure 30. Mapping the Class Definition of DS_C_ENTRY_INFO_SELECTION

A typical Directory Service operation, such as a read operation (**ds_read()**), requires the *entry-information-selection* parameter to specify to the Directory Service the information to be extracted from the directory entry. This *entry-information-selection* (a **DS_C_ENTRY_INFO_SELECTION** OM object) parameter is built by the application program as a public object (“Public Objects” on page 102 describes how to create a public object), and is included as a parameter to the **ds_read()** function call, as shown in the following code fragment from **example.c** :

```

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

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

```

Object Identifiers

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

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

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

The following code fragment from the `xdsbdcp.h` (the Basic Directory Contents Package) header file contains the symbolic constant **OMP_O_DS_A_COUNTRY_NAME**:

```

#ifndef dsP_attributeType /* joint-iso-ccitt(2) ds(5) attributeType(4) */
#define dsP_attributeType(X) ("5504" ##X)
#endif

#define OMP_O_DS_A_COUNTRY_NAME dsP_attributeType(06)

```

It resolves to 2.5.4.6, which is the object identifier value for Country Name attribute type as defined in the directory standard. Note that the "dotted decimal" and hexadecimal forms of the object identifier are slightly different. This is due to the definition of the basic encoding rules for object identifiers (ISO 8825).

C Naming Conventions

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

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

- XDS API function names are specified entirely in lowercase letters, and are prefixed by **ds_** (for example, **ds_read()**).
- XOM API Function names are specified entirely in lowercase letters, and are prefixed by **om_** (for example, **om_get()**).
- C function parameters are derived from the parameter and result names and are specified entirely in lowercase letters. In addition, the names of results have **_return** added as a suffix (for example, **operation_status_return**).
- OM class names are specified entirely in uppercase letters, and are prefixed by **DS_C_** and **MH_C_** (for example, **DS_C_AVA**).
- OM attribute names are specified entirely in uppercase letters, and are prefixed by **DS_** and **MH_** (for example, **DS_RDNS**).
- OM syntax names are specified entirely in uppercase letters, and are prefixed by **OM_S_** (for example, **OM_S_PRINTABLE_STRING**).
- Directory class names are specified entirely in uppercase letters, and are prefixed by **DS_O** (for example, **DS_O_ORG_PERSON**).
- Directory attribute names are specified entirely in uppercase letters, and are prefixed by **DS_A** (for example, **DS_A_COUNTRY_NAME**).
- Errors are treated as a special case. Constants that are the possible values of the OM attribute **DS_PROBLEM** of a subclass of the OM class **DS_C_ERROR** are specified entirely in uppercase letters, and are prefixed by **DS_E_** (for example, **DS_E_BAD_CLASS**).
- The constants in the Value Length and Value Number columns of the OM class definition tables are also assigned identifiers. Where the upper limit in one of these columns is *not* 1 (one), it is given a name consisting of the OM attribute name, prefixed by **DS_VL_** for value length, or **DS_VN_** for value numbers.
- The sequence of octets for each object identifier is also assigned an identifier for internal use by certain OM macros. These identifiers are all uppercase and are prefixed by **OMP_O_**.

Table 14 and Table 15 on page 102 summarize the XDS and XOM naming conventions.

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

Table 15. C Naming Conventions for XOM

Item	Prefix
Data type	OM_
Data value	OM_
Data value (Class)	OM_C_
Data value (Syntax)	OM_S_
Data value component (Structure member)	None
Function	om_
Function argument	None
Function result	None
Macro	OM_
Reserved for use by implementers	OMP
Reserved for use by implementers	omP
Reserved for proprietary extension	omX
Reserved for proprietary extension	OMX

Public Objects

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

XDS functions that perform directory operations, such as **ds_read()**, require *public* and/or *private* objects as input parameters. Typically, a public object is generated by an application program and contains the information required to access a target directory object. This information includes the AVAs and RDNs that make up a distinguished name of an entry in the directory. However, an application program can also generate a private object. Private objects are described in “Private Objects” on page 111.

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

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

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

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

Figure 31 on page 103 shows the representation of a public object in a descriptor list. The first descriptor in the list denotes the object’s OM class; the last descriptor is a **NULL** descriptor that signals

the end of the list of OM attributes. Between the first and the last descriptor are the descriptors for the OM attributes of the object.

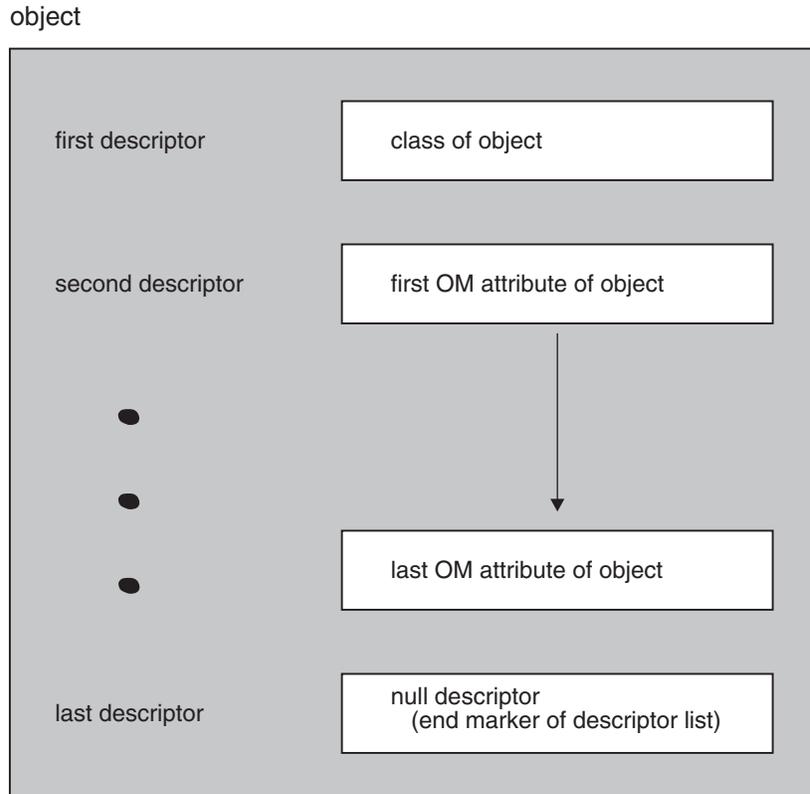


Figure 31. A Representation of a Public Object Using a Descriptor List

For example, the following represents the public object **Country** in **example.c**:

```

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

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

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

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

The third descriptor specifies the attribute value of the AVA of an object entry in the directory. The **OM_OID_DESC** macro is not used here because **OM_OID_DESC** is only used to initialize values having **OM_S_OBJECT_IDENTIFIER_STRING** syntax. The OM attribute type is **DS_ATTRIBUTE_VALUES**, the syntax is **OM_S_PRINTABLE_STRING**, and the value is **US**. The **OM_STRING** macro creates a data

value for a string data type (data type **OM_string**), in this case **OM_S_PRINTABLE_STRING**. A string is specified in terms of its length or whether or not it terminates with a **NULL**. (The **OM_STRING** macro is described in “The OM_STRING Macro” on page 135.)

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

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

OM_NULL_DESCRIPTOR has OM attribute type **OM_NO_MORE_TYPES**, syntax **OM_S_NO_MORE_SYNTAXES**, and value **OM_ELEMENTS_UNSPECIFIED**.

Figure 32 shows the composition of a descriptor list representing a public object.

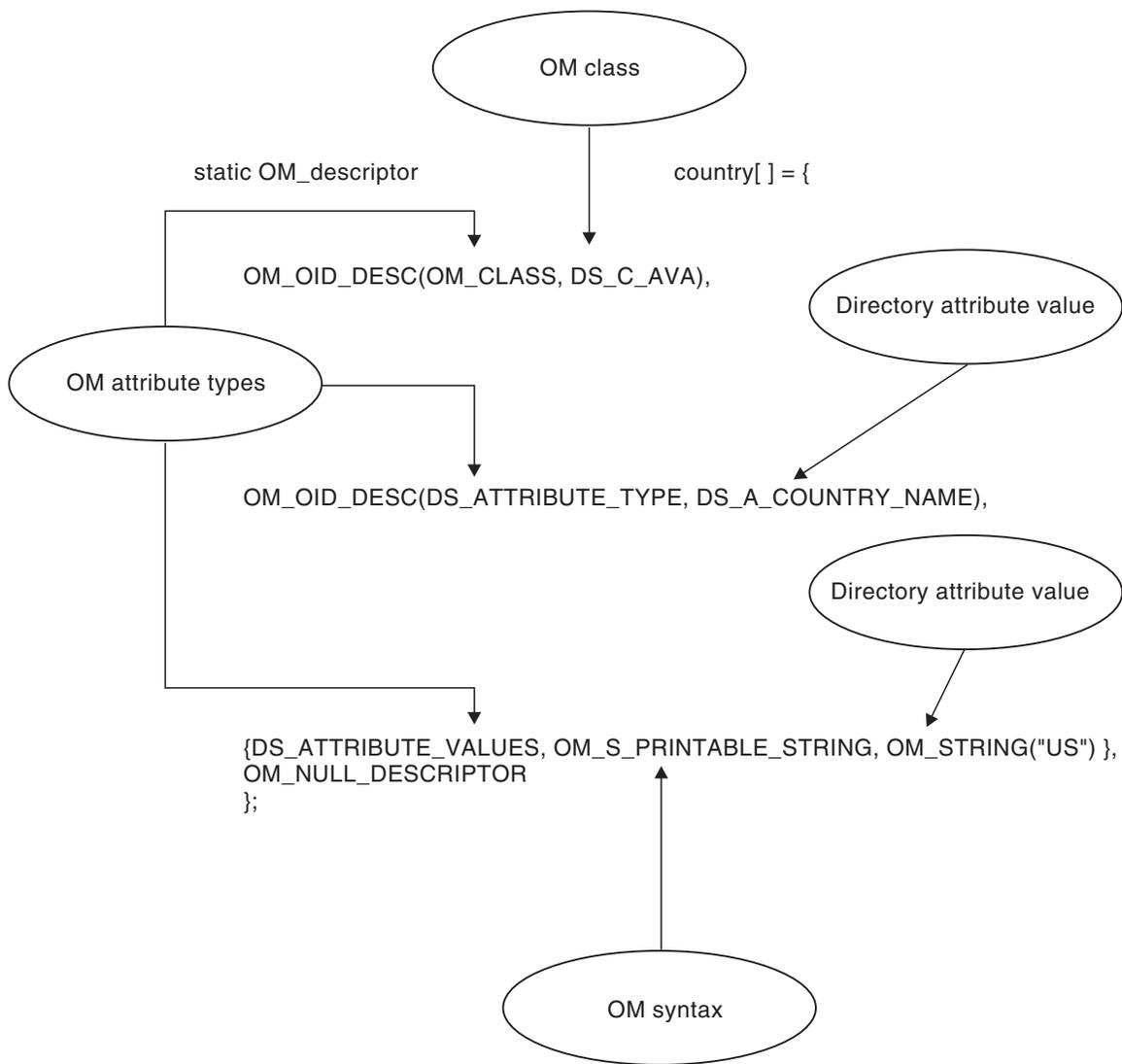
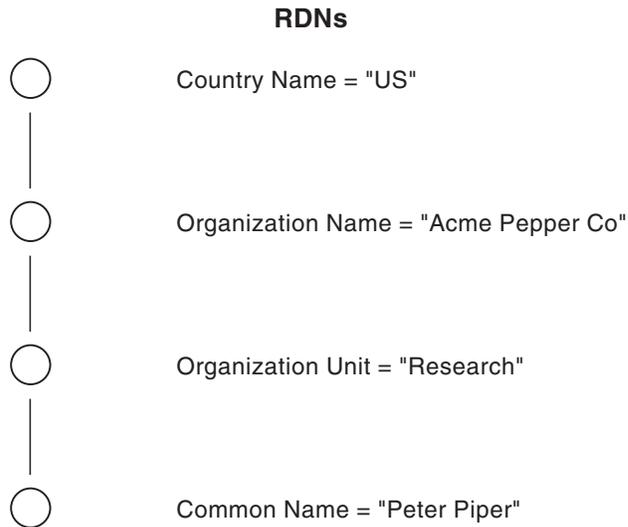


Figure 32. A Descriptor List for the Public Object: country

Building the Distinguished Name as a Public Object: Recall that RDNs are built from AVAs and a distinguished name is built from a series of RDNs. In a typical application program, several AVAs are defined in descriptor lists as public objects. These public objects are incorporated into descriptor lists that represent corresponding RDNs. Finally, the RDNs are incorporated into one descriptor list that represents the distinguished name of an object in the directory as shown in Figure 33. This descriptor list is included as one of the input parameters to a Directory Service function.



Distinguished Name = {C=US, O=Acme Pepper Co, OU=Research, CN=Peter Piper}

Figure 33. The Distinguished Name of Peter Piper in the DIT

Note: The above diagram shows how **example.c** works with GDS. To work with CDS, if your cell name is {C=US,O=Acme Pepper Co,OU=Research}, you need to change the RDN *CN="Peter Piper"* to be typeless. The **example.c** program used the typeless RDN form.

The following simplified code fragment from **example.c** (with the RDN with value "Peter Piper" changed to be typeless and a CDS directory entry named "Phone Book" added in order to access the CDS directory) shows how a distinguished name is built as a public object. The public object is the *name* parameter for a subsequent read operation call to the directory. The representation of a distinguished name in the DIT is shown in Figure 33.

The first section of code defines the five AVAs. These AVAs inform the Directory Service that the attribute values in the distinguished name of "Peter Piper" are valid and can therefore be read from the directory. The country name is "US," the organization name is "Acme Pepper Co," the organizational unit name is "Research," the CDS directory is "Phone Book," and the CDS object name is "Peter Piper."

Note: If the DCE cell name is a DNS type of name, then only three AVAs are required: the cell name, the CDS directory "Phone Book" and the CDS object name "Peter Piper."

```

/*
 * Public Object ("Descriptor List") for Name parameter to
 * ds_read( ).
 * Build the Distinguished-Name of Peter Piper
 */
static OM_descriptor      country[] = {
    OM_OID_DESC(OM_CLASS, DS_C_AVA),
    OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_COUNTRY_NAME),
    { DS_ATTRIBUTE_VALUES,OM_S_PRINTABLE_STRING | OM_S_LOCAL_STRING,OM_STRING("US") },
    OM_NULL_DESCRIPTOR
};
static OM_descriptor      organization[] = {
    OM_OID_DESC(OM_CLASS, DS_C_AVA),
    OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_ORG_NAME),
    { DS_ATTRIBUTE_VALUES,OM_S_PRINTABLE_STRING | OM_S_LOCAL_STRING,OM_STRING("Acme Pepper Co") },
    OM_NULL_DESCRIPTOR
};
static OM_descriptor      organizational_unit[] = {
    OM_OID_DESC(OM_CLASS, DS_C_AVA),
    OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_ORG_UNIT_NAME),
    { DS_ATTRIBUTE_VALUES,OM_S_PRINTABLE_STRING | OM_S_LOCAL_STRING,OM_STRING("Research") },
    OM_NULL_DESCRIPTOR
};
static OM_descriptor      directory_name[] = {
    OM_OID_DESC(OM_CLASS, DS_C_AVA),
    OM_OID_DESC(DS_ATTRIBUTE_TYPE, DSX_TYPELESS_RDN),
    { DS_ATTRIBUTE_VALUES,OM_S_PRINTABLE_STRING | OM_S_LOCAL_STRING,OM_STRING("Phone Book") },
    OM_NULL_DESCRIPTOR
};
static OM_descriptor      common_name[] = {
    OM_OID_DESC(OM_CLASS, DS_C_AVA),
    OM_OID_DESC(DS_ATTRIBUTE_TYPE, DSX_TYPELESS_RDN),
    { DS_ATTRIBUTE_VALUES,OM_S_PRINTABLE_STRING | OM_S_LOCAL_STRING,OM_STRING("Peter Piper") },
    OM_NULL_DESCRIPTOR
};

```

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

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

```

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

```

```

};
static OM_descriptor      rdn3[] = {
    OM_OID_DESC(OM_CLASS, DS_C_DS_RDN),
    { DS_AVAS, OM_S_OBJECT, { 0, organizational_unit } },
    OM_NULL_DESCRIPTOR
};
static OM_descriptor      rdn4[] = {
    OM_OID_DESC(OM_CLASS, DS_C_DS_RDN),
    { DS_AVAS, OM_S_OBJECT, { 0, directory_name } },
    OM_NULL_DESCRIPTOR
};
static OM_descriptor      rdn5[] = {
    OM_OID_DESC(OM_CLASS, DS_C_DS_RDN),
    { DS_AVAS, OM_S_OBJECT, { 0, object_name } },
    OM_NULL_DESCRIPTOR
};

```

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

```

OM_descriptor      name[] = {
    OM_OID_DESC(OM_CLASS, DS_C_DS_DN),
    { DS_RDNS, OM_S_OBJECT, { 0, rdn1 } },
    { DS_RDNS, OM_S_OBJECT, { 0, rdn2 } },
    { DS_RDNS, OM_S_OBJECT, { 0, rdn3 } },
    { DS_RDNS, OM_S_OBJECT, { 0, rdn4 } },
    { DS_RDNS, OM_S_OBJECT, { 0, rdn5 } },
    OM_NULL_DESCRIPTOR
};

```

In summary, the distinguished name for **Peter Piper** is stored in the array of descriptors called **name**, which is composed of three nested levels of arrays of descriptors (see Figure 34 on page 108). The definitions for the AVAs are at the innermost level, the definitions for RDNs are at the next level up, and the distinguished name is at the top level.

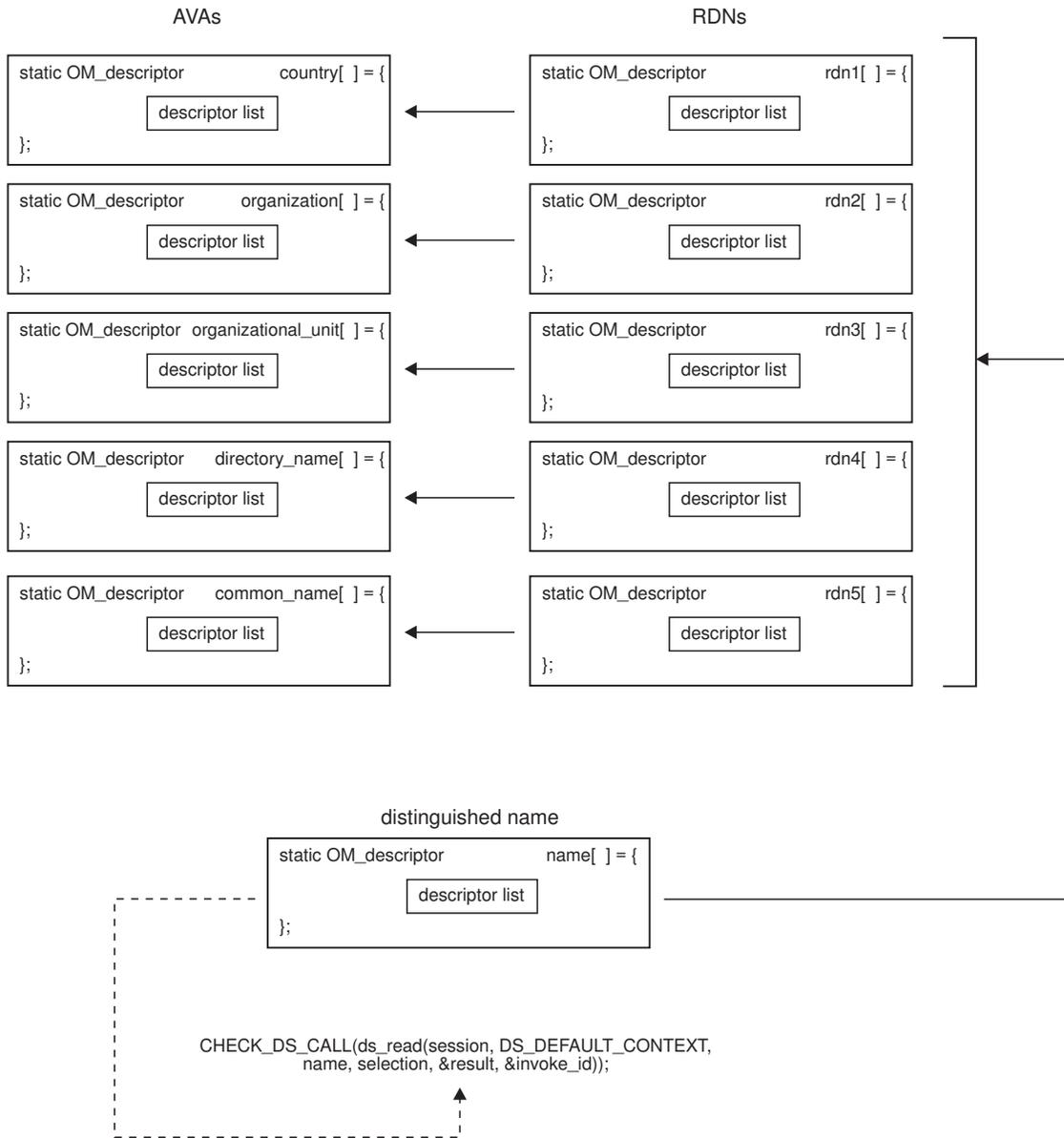


Figure 34. Building a Distinguished Name

Figure 35 on page 109 shows a more general view of the structure of a distinguished name.

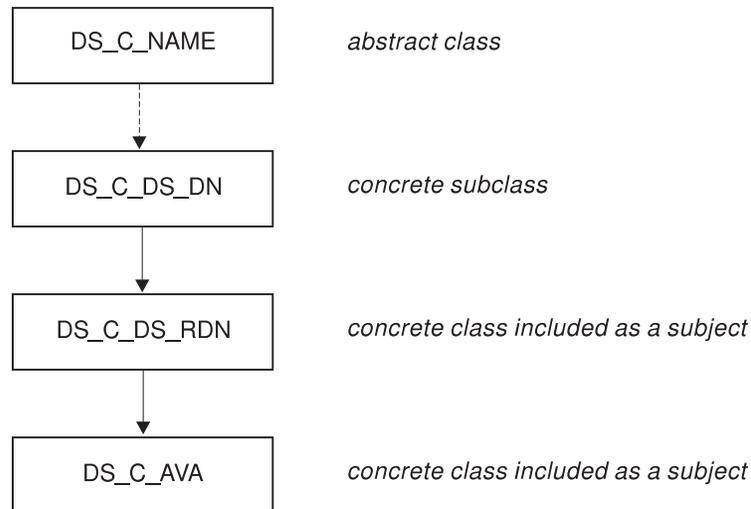


Figure 35. A Simplified View of the Structure of a Distinguished Name

The **name** descriptor defines a public object that is provided as the *name* parameter required by the XDS API read function call, **ds_read()**, as follows. (XDS API function calls are described in detail in Chapter 6, “XDS Programming” on page 137.)

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

The result of the **ds_read()** function call is in a private implementation-specific format; it is stored in a workspace and pointed to by *result*. The application program must use XOM function calls (described in “OM Function Calls” on page 128) to interpret the data and extract the information. This extraction process involves uncovering the nested data structures in a series of XOM function calls.

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

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

For example, Figure 36 on page 110 shows how the read request described in the previous example returns a pointer to an encoded data structure stored in **result**. This encoded data structure, referred to as a *private object* (described in the next section), is one of the input parameters to **om_get()**. The **om_get()** function provides a pointer to a public object (in this case, **entry**) as an output parameter. The public object is a data structure that has been interpreted by **om_get()** and is accessible by the application program (the client). The information requested by the application in the read request is contained in the output parameter **entry**.

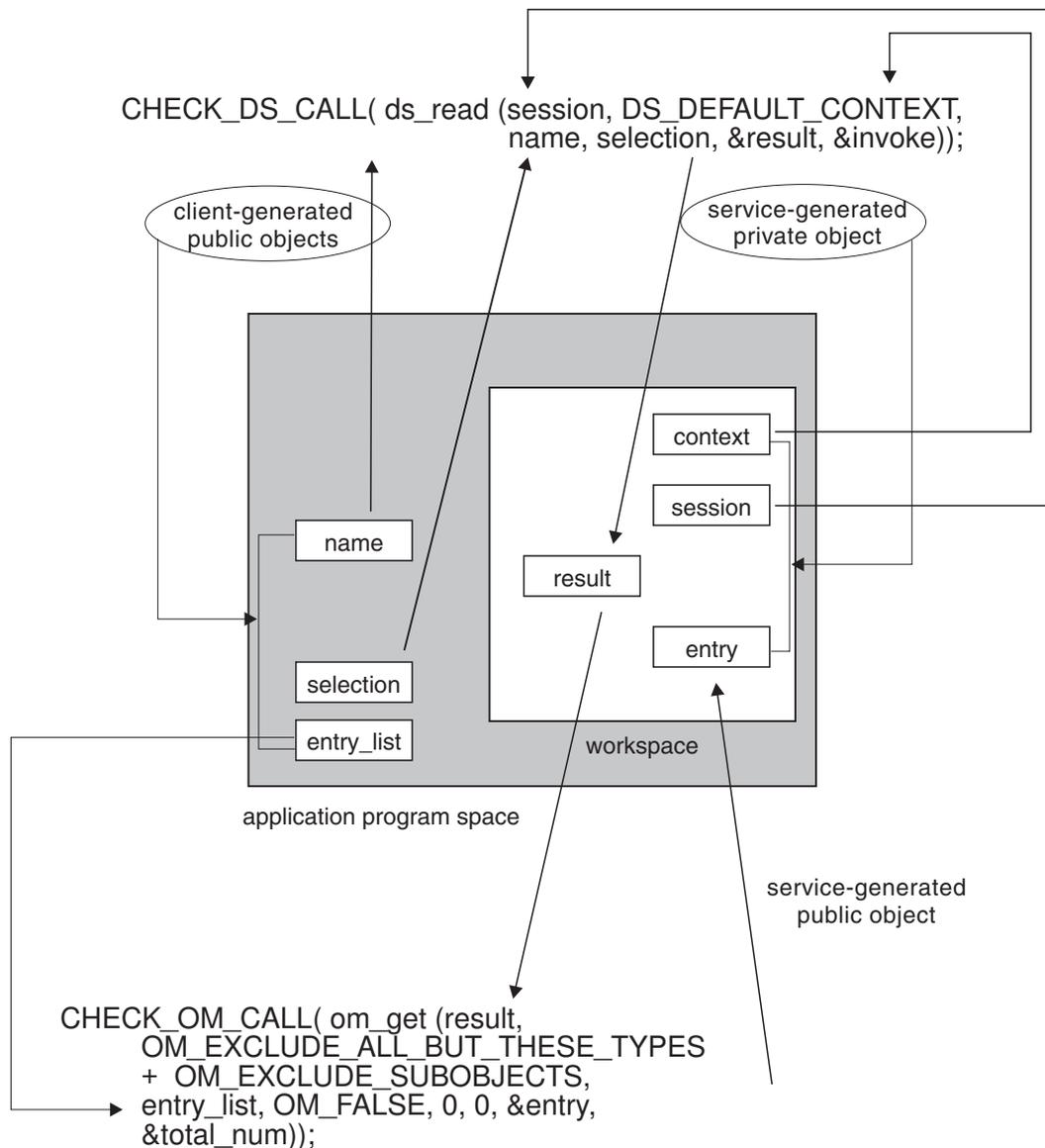


Figure 36. Client-Generated and Service-Generated Objects

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

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

```

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

```

Private Objects

Private objects are created dynamically by the service interface. In Figure 36 on page 110, the **ds_read()** function returns a pointer to the data structure *result* in the workspace. This service-generated data structure is a private object in a private implementation-specific format, which requires a call to **om_get()** to interpret the data. A private object is one of the required input parameters to XOM API functions (such as **om_get()**), as shown in Figure 36 on page 110. Private objects are always service generated.

Table 16 compares private and public objects.

Table 16. Comparison of Private and Public Objects

Private	Public
Representation is implementation specific.	Representation is defined in the API specification.
Not directly addressable by client.	Directly addressable by client.
Manipulated by client using OM functions.	Manipulated by client using programming constructs.
Created in storage provided by the service.	Is <i>service generated</i> if created by the service. Is <i>client generated</i> if created by the client, in storage provided by the client.
Cannot be modified by the client directly (except through the service interface).	If <i>client generated</i> , can be modified directly by the client. If <i>service generated</i> , cannot be modified directly by the client (except through the service interface).
Storage allocated and released by the service.	Storage allocated and released by the service if <i>service generated</i> ; storage allocated and released if <i>client generated</i> .

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

Object Classes

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

Later, this section shows how OM classes are organized into groups of OM classes, called packages, that support some aspect of the Directory Service.

OM Class Hierarchy and Inheritance Properties: OM Classes are related to each other in a tree hierarchy whose root is a special OM class called **OM_C_OBJECT**. Each of the other OM classes is the immediate subclass of precisely one other OM class. This tree structure is known as the *OM class hierarchy*. It is important because of the property of inheritance. The OM class hierarchy is defined by the XDS/XOM. DCE implements this hierarchy for GDS and adds its own set of OM classes defined in the GDS Package.

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

For example, as shown in Figure 37, **DS_C_ENTRY_INFO_SELECTION** inherits its OM attributes from its superclass **OM_C_OBJECT**. The OM attributes, **DS_ALL_ATTRIBUTES**, **DS_ATTRIBUTES_SELECTED**, and **DS_INFO_TYPE** are attributes specific to OM class **DS_C_ENTRY_INFO_SELECTION**.

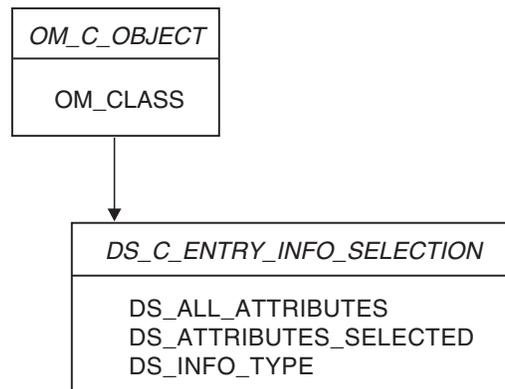


Figure 37. The OM Class **DS_C_ENTRY_INFO_SELECTION**

Another important point about OM class inheritance is that an instance of an OM class is also considered to be an instance of each of its superclasses, and can appear wherever the interface requires an instance of any of those superclasses.

For example, **DS_C_DS_DN** is a subclass of **DS_C_NAME**. Everywhere in an application program where **DS_C_NAME** is expected at the interface (a parameter to a **ds_read()**, for example), you can supply **DS_C_DS_DN**.

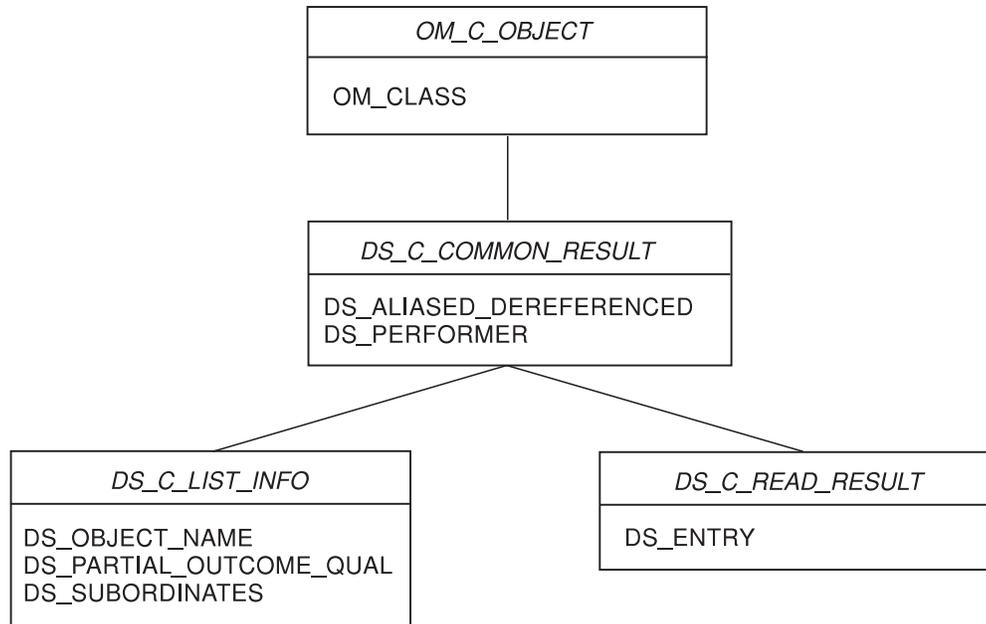
Abstract and Concrete Classes: OM Classes are defined as *abstract* or *concrete*.

An abstract OM class is an OM class in which instances are not permitted. An abstract OM class can be defined so that subclasses can share a common set of OM attributes between them.

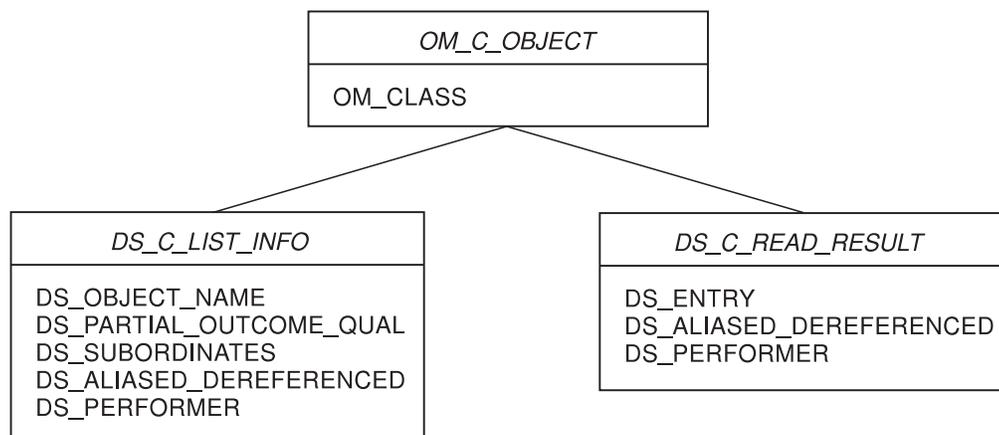
In contrast to abstract OM classes, instances of OM concrete classes are permitted. However, the definition of each OM concrete class can include the restriction that a client not be allowed to create instances of that OM class.

For example, consider two alternative means of defining the OM classes use in XDS: **DS_C_LIST_INFO** and **DS_C_READ_RESULT**. **DS_C_LIST_INFO** and **DS_C_READ_RESULT** are subclasses of the abstract OM class **DS_C_COMMON_RESULT**.

Figure 38 shows the relationship of **DS_C_LIST_INFO** and **DS_C_READ_RESULT** when the abstract OM class **DS_C_COMMON_RESULT** is defined and when it is not defined. It demonstrates that the presence of an abstract OM class enables the programmer to develop applications that process information more efficiently.



DS_C_LIST_INFO and DS_C_READ_RESULT with the DS_C_COMMON_RESULT abstract class defined



DS_C_LIST_INFO and DS_C_READ_RESULT without the DS_C_COMMON_RESULT abstract class defined

Figure 38. A Comparison of Two Classes With and Without an Abstract OM Class

The following list contains the hierarchy of concrete and abstract OM classes in the Directory Service Package. Abstract OM classes are shown in italics. The indentation shows the class hierarchy; for example the abstract class *OM_C_OBJECT* is a superclass of the abstract class

DS_C_COMMON_RESULTS, which in turn is a superclass of the concrete class **DS_C_COMPARE_RESULT**.

OM_C_OBJECT

- **DS_C_ACCESS_POINT**
- *DS_C_ADDRESS*
 - **DS_C_PRESENTATION_ADDRESS**
- **DS_C_ATTRIBUTE**
 - **DS_C_AVA**
 - **DS_C_ENTRY_MOD**
 - **DS_C_FILTER_ITEM**
- **DS_C_ATTRIBUTE_ERROR**
- **DS_C_ATTRIBUTE_LIST**
 - **DS_C_ENTRY_INFO**
- *DS_C_COMMON_RESULTS*
 - **DS_C_COMPARE_RESULT**
 - **DS_C_LIST_INFO**
 - **DS_C_READ_RESULT**
 - **DS_C_SEARCH_INFO**
- **DS_C_CONTEXT**
- **DS_C_CONTINUATION_REF**
 - **DS_C_REFERRAL**
- **DS_C_ENTRY_INFO_SELECTION**
- **DS_C_ENTRY_MOD_LIST**
- *DS_C_ERROR*
 - **DS_C_ABANDON_FAILED**
 - **DS_C_ATTRIBUTE_PROBLEM**
 - **DS_C_COMMUNICATIONS_ERROR**
 - **DS_C_LIBRARY_ERROR**
 - **DS_C_NAME_ERROR**
 - **DS_C_SECURITY_ERROR**
 - **DS_C_SERVICE_ERROR**
 - **DS_C_SYSTEM_ERROR**
 - **DS_C_UPDATE_ERROR**
- **DS_C_EXT**
- **DS_C_FILTER**
- **DS_C_LIST_INFO_ITEM**
- **DS_C_LIST_RESULT**

- *DS_C_NAME*
 - **DS_C_DS_DN**
- **DS_C_OPERATION_PROGRESS**
- **DS_C_PARTIAL_OUTCOME_QUAL**
- *DS_C_RELATIVE_NAME*
 - **DS_C_DS_RDN**
- **DS_C_SEARCH_RESULT**
- **DS_C_SESSION**

In summary, an OM class is defined with the following elements:

- OM Class Name (indicated by an object identifier)
- Identity of its immediate superclass
- Definitions of the OM attribute types specific to the OM class
- Indication whether the OM class is abstract or concrete
- Constraints on the OM attributes.

A complete description of OM classes, OM attributes, syntaxes, and values that are defined for XDS and XOM APIs are described elsewhere in the GDS chapters, and in the XDS/XOM supplementary information. Tables and textual descriptions, such as the one shown in Figure 39 on page 116 for the concrete OM class **DS_C_ATTRIBUTE**, are provided for each OM class in these chapters.

DS_C_ATTRIBUTE

Class name

Description of the class including an indication if it is an abstract class

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

Indicates superclasses

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

OM Attributes of a DS_C_ENTRY_INFO_SELECTION

OM Attribute	Value Syntax	Value Length	Value No.	Value Initially
DS_ATTRIBUTE_TYPE	String(OM_S_OBJECT_IDENTIFIER_STRING)	-	1	-
DS_ATTRIBUTE_VALUES	any	-	0 or more	-

Table showing values of syntax, length, number of values, and initial value

➤ **DS_ATTRIBUTE_TYPE**

Description of attributes and listing of attribute values

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

➤ **DS_ATTRIBUTE_VALUES**

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

Note: A directory attribute must always have at least one value, although it is acceptable for an instance of this OM class to have no values.

Figure 39. A Complete Description of the Concrete OM Class DS_C_ATTRIBUTE

The table shown in Figure 39 provides information under the following headings:

- OM Attribute
The name of each of the OM attributes
- Value Syntax
The syntaxes of each of the OM attribute's values
- Value Length
Any constraints on the number of bits, octets or bytes, or characters in each value that is a string
- Value Number
Any constraints on the number of values
- Value Initially
Any value with which the OM attribute can be initialized.

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

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

An object passed as a parameter to an XOM and XDS function call needs to meet a minimum set of conditions:

- The type of each OM attribute must be specific to the object's OM class or one of its superclasses.
- The number of values of each OM attribute must be within OM class limits.
- The syntax of each value must be among those the OM class permits.
- The number of bits, octets, or characters in each string value must be within OM class limits.

Packages

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

A package is uniquely identified by an ASN.1 object identifier. DCE XDS API supports four packages of which one is mandatory and three are optional:

- Directory Service Package (mandatory)
- Basic Directory Contents Package (optional)
- Strong Authentication Package (optional)
- Global Directory Service Extension Package (optional)
- MHS Directory User package (optional).

Note: All of the above packages included in the API; however, the objects and attributes supported by the GDS extension package and the MHS Directory User package are complex and suited for GDS and mail objects in GDS. z/OS DCE does not support GDS.

The Directory Service Package

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

The object identifiers for specific packages are defined in header files that are part of the XDS API and XOM API. An object identifier consists of a string of integers. The header files include **#define** preprocessor statements that assign names to these constants to make them more readable. These assignments alleviate the application programmer from the burden of maintaining these strings of integers.

For example, the object identifiers for the Directory Service Package are defined in **xds.h**. The **xds.h** header file contains OM class and OM attribute names, OM object constants, and defines prototypes for XDS API functions, as shown in the following code fragment from **xds.h**:

```

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

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

```

A **ds_version()** function call must be included within an application program to negotiate the optional features (packages) with the Directory Service. The first step is to build an array of object identifiers for the optional packages to be negotiated (the Basic Directory Contents Package) as shown in the following code fragment from the **teldir.c** application:

```

/*
 * To identify which packages we need for this program. We only need
 * the basic package because we are not doing anything fancy with
 * session parameters, etc.
 */
DS_feature featureList[] = {
    { OM_STRING(OMP_O_DS_BASIC_DIR_CONTENTS_PKG), OM_TRUE },
    { 0 }
};

```

The **OM_STRING** macro is provided for creating a data value of data type **OM_string** for octet strings and characters. XOM API macros are described in “XOM API Macros” on page 133.

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

```

/* STEP 3
 *
 * Pull in the packages that contain the XDS features we need.
 */
dsStatus = ds_version( featureList, xdsWorkspace );
if( dsStatus != DS_SUCCESS )
    handleDSError( "ds_version()", dsStatus );

```

For information about workspaces, see the section entitled “Workspaces” on page 120.

The Basic Directory Contents Package

The Basic Directory Contents Package contains the object identifier definition of directory classes and attribute types as defined by the X.500 Standard. These definitions allow the creation and maintenance of directory entries for a number of common objects so that the representation of all such objects is the same throughout the directory. Also included are the definitions of the OM classes and OM attributes required to support the directory attribute types. Chapter 12, “Basic Directory Contents Package” on page 275 describes the Basic Directory Contents Package in detail.

The object identifier associated with the Basic Directory Contents Package is shown in the following code fragment from the **xdsbdcp.h** header file:

```

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

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

```

The Strong Authentication Package

The Strong Authentication Package contains the object identifier definition of directory classes and attribute types as defined by the X.500 standard for security purposes. Also included are the definitions of the OM classes and OM attributes required to support these security attribute types. Chapter 13, “Strong Authentication Package” on page 289 describes the Strong Authentication Package in detail.

The object identifier associated with the Strong Authentication Package is shown in the following code fragment from the **xdssap.h** header file:

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

#define OMP_DS_STRONG_AUTHENT_PKG "\x2B\x0C\x02\x87\x73\x1C\x02"
```

The Global Directory Service Package

The Global Directory Service Package contains the definition of a DCE extension to the XDS API. It contains the definitions of OM classes, OM attributes, and syntaxes to support extended functionality specific to DCE. Chapter 15, “Global Directory Service Package” on page 313 describes the GDS Package in detail.

Note: The XDS/XOM API supplied with z/OS DCE cannot access an X.500 directory. These attribute definitions are provided in case applications wish to use them in CDS entries.

The following code fragment from the **xdsgds.h** header file shows the object identifier for the GDS Package:

```
/* GDS package object identifier */
/* { iso(1) identifier-organization(3) icd-ecma(12) member-company (2)
 *   siemens-units(1107) sni(1) directory(3) xds-api(100) gdsp(1) } */

#define OMP_0_DSX_GDS_PKG "\x2B\x0C\x02\x88\x53\x01\x03\x64\x01"
```

The XDS/XOM supplementary information section describes in detail the attributes and data types that make up the OM and Directory classes defined in the XDS API packages. Chapter 7, “Example Application Programs” on page 159 examines in detail how these packages are used in developing the sample application programs.

The MHS Directory User Package

The Message Handling System Directory User Package contains definitions to support the use of the directory in accordance with the standard X.400 (1988) User Agents and Message Transfer Agents (MTAs) for name resolution, Distribution List expansion, and capability assessment. The definitions are based on the attribute types and syntaxes specified in X.402, Annex A. The definitions of the OM classes and OM attributes required to support these MHS attribute types are also included with this package. Chapter 14, “MHS Directory User Package” on page 297 describes the MHS Directory User Package in detail.

Note: The XDS/XOM API supplied with z/OS DCE cannot access an X.500 directory. These attribute definitions are provided in case applications wish to use them in CDS entries.

The object identifier associated with the MHS Directory User Package is shown in the following code fragment from the **xdsmdup.h** header file:

```

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

#define OMP_DS_MHS_DIR_USER_PKG "\x2B\x0C\x02\x87\x73\x1C\x03"

```

Part 4, “XDS/XOM Supplementary Information” on page 231 describes in detail the attributes and data types that make up the OM and directory classes defined in the XDS API packages. Chapter 7, “Example Application Programs” on page 159 examines in detail how these packages are used in developing the sample application programs.

Package Closure

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

Workspaces

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

Recall that private objects are returned by a service to a workspace in private implementation-specific format. Using the OM functions (described in “OM Function Calls” on page 128), the data can be extracted from the private object for further program processing.

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

The programs in Chapter 7, “Example Application Programs” on page 159 demonstrate how to initialize and shutdown a workspace. The XDS functions, **ds_initialize()** and **ds_shutdown()**, are described in detail in “The ds_initialize() Function Call” on page 138 and “The ds_shutdown() Function Call” on page 140 respectively.

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

For example, some of these operations in an application may require involvement with GDS security, ACLs, or the DUA cache. Therefore, in addition to the basic packages provided by the Directory Service APIs, the workspace would have to support the GDSP package. The following code fragment demonstrates how an application initializes a workspace and negotiates the packages to be associated with that workspace:

```

/* Build up an array of object identifiers for the optional */
/* packages to be negotiated.                               */

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

CHECK_DS_CALL((OM_object) !(workspace = ds_initialize));

CHECK_DS_CALL(ds_version(bdcg_package, workspace));

```

Storage Management

An object occupies storage. The storage occupied by a public object is allocated by the client, and is, therefore, directly accessible by the client and can be released by the client. (Note that a public object created using the XOM function call **om_get()** on a private object should be released using the **om_delete** XOM function.) The storage occupied by a private object is not accessible by the client and must be managed indirectly using XOM function calls.

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

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

```

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

```

One of the input parameters to the **ds_read()** function call is **name**. The **name** parameter is a public object created by the application from a series of nested data structures (RDNs and AVAs) to represent the distinguished name which in the example contains **Peter Piper**. The **ds_read()** call returns the pointer to a private object, **result**, deposited in the workspace by the service.

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

```

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

CHECK_OM_CALL( om_get(result,
    OM_EXCLUDE_ALL_BUT_THESE_TYPES | OM_EXCLUDE_SUBOBJECTS,
    entry_list, OM_TRUE, 0, 0, &entry, &total_num));

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

CHECK_OM_CALL( om_get(attributes->value.object.object,
    OM_EXCLUDE_ALL_BUT_THESE_TYPES | OM_EXCLUDE_SUBOBJECTS,
    telephone_list, OM_TRUE, 0, 0, &telephones, &total_num));

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

```

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

OM Syntaxes for Attribute Values

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

Some syntaxes are described in the standard in terms of syntax templates. A syntax template defines a group of related syntaxes. The syntax templates that are defined are:

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

Enumerated Types

An OM attribute with syntax template Enum(*) is an enumerated type (**OM_S_ENUMERATION**) and has a set of values associated with that OM attribute. For example, one of the OM attributes of the OM class **DS_C_ENTRY_INFO_SELECTION** is **DS_INFO_TYPE**. **DS_INFO_TYPE** is listed in the OM attribute table for **DS_C_ENTRY_INFO_SELECTION** in Chapter 11, “XDS Class Definitions” on page 241 as having a value syntax of Enum(**DS_Information_Type**), as shown in Table 17. **DS_INFO_TYPE** takes one of the following values:

- **DS_TYPES_ONLY**
- **DS_TYPES_AND_VALUES**.

Table 17. Description of an OM Attribute Using Syntax Enum(*). OM Attributes of DS_C_Entry_Info_SELECTION

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_ALL_ATTRIBUTES	OM_S_BOOLEAN	-	1	OM_TRUE
DS_ATTRIBUTES_SELECTED	String(OM_S_OBJECT_IDENTIFIER_STRING)	-	0 or more	-
DS_INFO_TYPE	Enum(DS_INFORMATION_TYPE)	-	1	DS_TYPES_AND_VALUES

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

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

Object Types

An OM attribute with syntax template Object(*) has **OM_S_OBJECT** as syntax and a subobject as a value. For example, one of the OM attributes of the OM class **DS_C_DS_DN** is **DS_RDNS**. **DS_RDNS** is listed in the OM attribute table for **DS_C_DS_DN** as having a value syntax of Object(**DS_C_DS_RDN**), as shown in Table 18.

Table 18. Description of an OM Attribute with Syntax Object(*). OM Attributes of DS_C_DS_DN

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_RDNS	Object(DS_C_DS_RDN)	-	0 or more	-

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

```
OM_descriptor      name[] = {
  OM_OID_DESC(OM_CLASS, DS_C_DS_DN),
  { DS_RDNS, OM_S_OBJECT, { 0, rdn1 } },
  { DS_RDNS, OM_S_OBJECT, { 0, rdn2 } },
  { DS_RDNS, OM_S_OBJECT, { 0, rdn3 } },
  { DS_RDNS, OM_S_OBJECT, { 0, rdn4 } },
  { DS_RDNS, OM_S_OBJECT, { 0, rdn5 } },
  OM_NULL_DESCRIPTOR
};
```

Strings

An OM attribute with syntax template `String(*)` specifies the string syntax of its value. A string is categorized as either a *bit string*, an *octet string*, or a *character string*. The bits of a bit string, the octets of an octet string, or the octets of a character string constitute the *elements* of the string. (Refer to Chapter 17, “Information Syntaxes” on page 325 for a list of the syntaxes that form the string group.)

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

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

For example, one of the OM attributes of the OM class **DS_C_ENTRY_INFO_SELECTION** is **DS_ATTRIBUTES_SELECTED**. **DS_ATTRIBUTES_SELECTED** is listed in the OM attribute table for **DS_C_ENTRY_INFO_SELECTION** as having a value syntax of `String(OM_S_OBJECT_IDENTIFIER_STRING)`, as shown in Table 17 on page 123.

Other Syntaxes

The other syntaxes are defined as follows:

- OM_S_BOOLEAN** A value of this syntax is a Boolean; that is, the value can be **OM_TRUE** or **OM_FALSE**.
- OM_S_INTEGER** A value of this syntax is a positive or negative integer.
- OM_S_NULL** The one value of this syntax is a valueless place holder.

Service Interface Data Types

The local variables within an application program that contain the parameters and results of XDS and XOM API function calls are declared using a standard set of data types. These data types are defined by **typedef** statements in the **xom.h** header files. Some of the more commonly used data types are described in the following subsections. A complete description of service interface data types is provided in Chapter 18, “XOM Service Interface” on page 331, and in *z/OS DCE Application Development Reference*.

The OM_descriptor Data Type

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

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

```
/* Descriptor */

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

OM_descriptor is made up of a series of nested data structures, as shown in Figure 40.

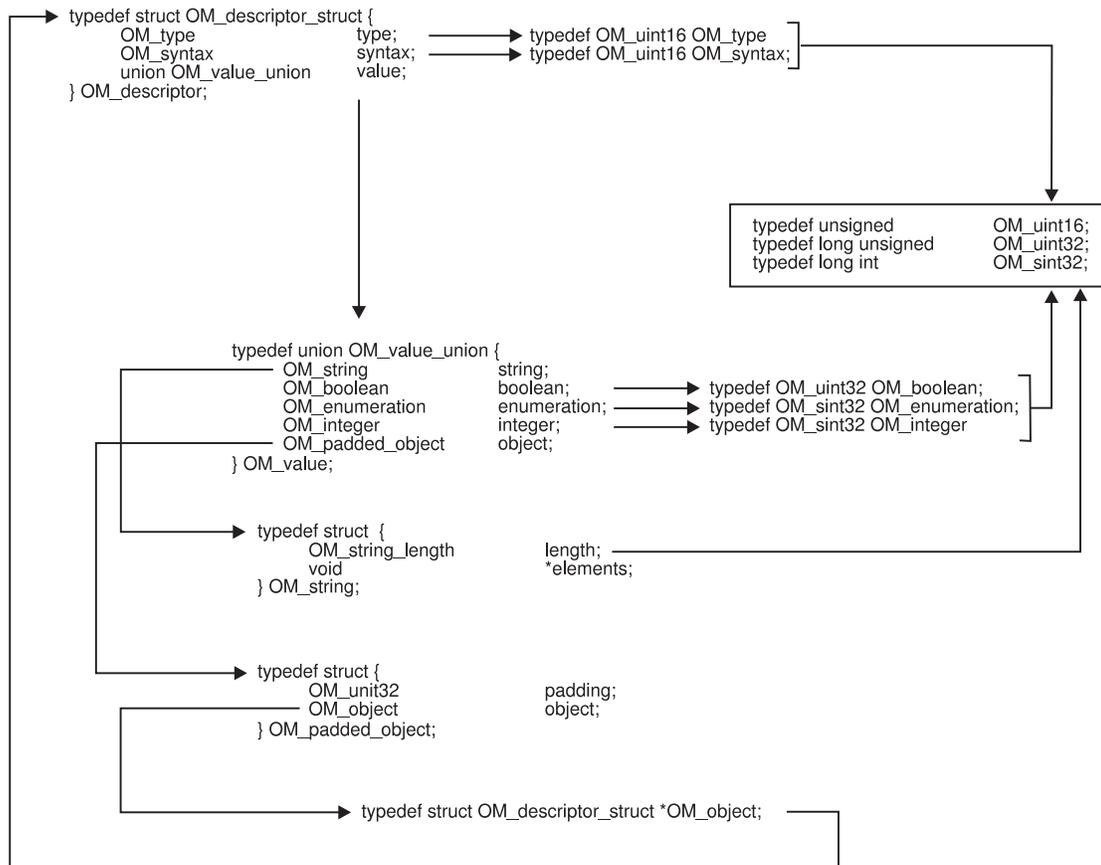


Figure 40. Data Type: `OM_descriptor_struct`

Figure 40 shows that **type** and **syntax** are integer constants for an OM attribute type and syntax, as shown in the following simplified code fragment from **example.c**:

```

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

```

The code fragment initializes four descriptors, as shown in Figure 41. The type and syntax evaluate to integers for all four descriptors.

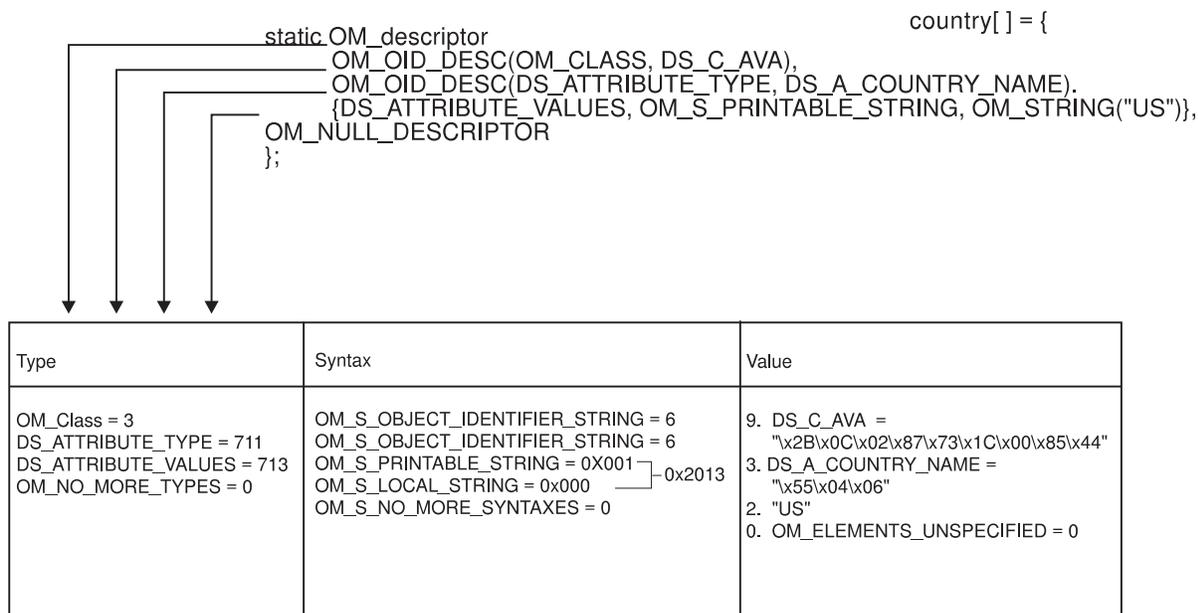


Figure 41. Initializing Descriptors

The **value** component is a little more complex. Figure 40 on page 125 shows that **value** is a union of **OM_value_union**. **OM_value_union** has five members: **string**, **boolean**, **enumeration**, **integer**, and **object**. The **boolean**, **enumeration**, and **integer** members have integer values. The **string** member contains a string of type **OM_string**, which is a structure composed of a length and a pointer to a string of characters. The **object** member is a structure of type **OM_padded_object** that points to another object nested below it. Many OM attributes have other objects as values. These subobjects, in turn, may have other subobjects, and so on.

For example, as shown in Figure 42 on page 127, the OM class **DS_C_READ_RESULT** has one OM attribute: **DS_ENTRY**. The syntax of **DS_ENTRY** is **OM_S_OBJECT** with a value of **DS_C_ENTRY_INFO**, indicating that it points to the subobject **DS_C_ENTRY_INFO**. **DS_C_ENTRY_INFO** has the OM attribute **DS_OBJECT_NAME** with the syntax **OM_S_OBJECT**, indicating that it points to the subobject **DS_C_NAME**.

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

Figure 42. An Object and a Subordinate Object

Data Types for XDS API Function Calls

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

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

The code fragment shows:

- The **ds_initialize** call returns a variable of type **OM_workspace** that contains a handle or pointer to a workspace.
- The **ds_bind()** call returns a pointer to a variable of type **OM_private_object**. The private object contains the session information required by all subsequent XDS API calls (except **ds_shutdown()**).
- The **ds_read()** call returns a pointer to the result of a directory read request in a variable of type **OM_private_object**.
- The error handling macros **CHECK_DS_CALL** and **CHECK_OM_CALL** (defined in **example.h** header file) use the data types **DS_status** and **OM_return_code**, respectively, as return values from XDS and XOM API function calls.

Data Types for XOM API Calls

The following code fragment from **example.c** shows how the datatypes are used to declare the variables that contain the input and output parameters for the XOM API function calls:

```

/*
 * variables to extract the telephone number(s)
 * */
OM_type          entry_list[]      = { DS_ENTRY, 0 };
OM_type          attributes_list[] = { DS_ATTRIBUTES, 0 };
OM_type          telephone_list[]  = { DS_ATTRIBUTE_VALUES, 0 };
OM_public_object entry;
OM_public_object attributes;
OM_public_object telephones;
OM_descriptor    * telephone;      /* current phone number */
OM_value_position total_num;       /* number of Attribute Descriptors */

```

The code fragment shows:

- The series of **om_get()** calls require a list of OM attribute types which identifies the types of OM attributes to be included in the operation. The variables *entry_list*, *attribute_list*, and *telephone_list* are declared as type **OM_type**.
- The series of **om_get()** calls provide as output pointers to variables of type **OM_public_object**. **om_get()** generates public objects that are accessible to the application program.
- Where the variable *total_num* is type **OM_value_position** and is used to hold the number of OM descriptors returned by **om_get()**.

Chapter 17, “Information Syntaxes” on page 325 contains detailed descriptions of all the data types defined by XOM API.

OM Function Calls

XOM API supports general-purpose OM functions defined by the X/Open standards body that allow an application program to manipulate objects in a workspace. “Summary of OM Function Calls” lists the OM function calls and gives a brief description of each. “Using the OM Function Calls” on page 129 illustrates the use of OM function calls using the **om_get()** call as an example.

Summary of OM Function Calls

The following list of XOM API function calls contains a brief description of each function. Refer to the *z/OS DCE Application Development Reference* for a detailed description of the input and output parameters, return codes, and usage of each function.

- **om_copy()**
Creates an independent copy of an existing private object and all of its subobjects into a specified workspace.
- **om_copy_value()**
Replaces an existing OM attribute value or inserts a new value into a target private object with a copy of an existing OM attribute value found in a source private object.
- **om_create()**
Creates a private object that is an instance of the specified OM class.
- **om_delete()**
Deletes a private or service-generated public object.
- **om_get()**

Creates a new public object that is an exact but independent copy of an existing private object; certain exclusions and/or syntax conversion may be requested for the copy.

- **om_instance()**

Tests to determine if an object is an instance of a specified OM class (includes the case when the object is a subclass of that OM class).

- **om_put()**

Places or replaces copies of the attribute values of the source private or public object into the target private object.

- **om_read()**

Reads a segment of a string attribute from a private object.

- **om_remove()**

Removes and discards values of an attribute of a private object.

- **om_write()**

Writes a segment of a string attribute to a private object.

- **om_encode()**

Not supported by DCE XOM API.

- **om_decode()**

Not supported by DCE XOM API.

Using the OM Function Calls

Most application programs require the use of a series of **om_get()** function calls to create service-generated public objects from which the program can extract requested information. This section uses the operation of **om_get()** as an example to describe how XOM API functions operate in general.

The following code fragment from **example.c** shows how a series of **om_get()** function calls extract a list of telephone numbers from a workspace. The **ds_read()** function call deposits the private object stored in **result** in the workspace and provides access to it by the pointer **&result**.

```
/*
 * extract the telephone number(s) of "name" from the result
 *
 * There are 4 stages:
 * (1) get the Entry-Information from the Read-Result.
 * (2) get the Attributes from the Entry-Information.
 * (3) get the list of phone numbers.
 * (4) scan the list and print each number.
 */
CHECK_OM_CALL( om_get(result,
    OM_EXCLUDE_ALL_BUT_THESE_TYPES | OM_EXCLUDE_SUBOBJECTS,
    entry_list, OM_TRUE, 0, 0, &entry, &total_num));

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

CHECK_OM_CALL( om_get(attributes->value.object.object,
    OM_EXCLUDE_ALL_BUT_THESE_TYPES | OM_EXCLUDE_SUBOBJECTS,
```

```

        telephone_list, OM_TRUE, 0, 0, &telephones, &total_num));

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

for (telephone = telephones; telephone->type != OM_NO_MORE_TYPES; telephone++)
{
    if (telephone->type != DS_ATTRIBUTE_VALUES)
    {
        fprintf(stderr, "malformed telephone number\n");
        exit(EXIT_FAILURE);
    }

    /*
     * copy and append '\0' to the telephone number, and then print
     */
    strncpy(telephone_str, telephone->value.string.elements,
            min(33, telephone->value.string.length));
    telephone_str[34] = '\0';
    printf("Telephone number: %s\n", telephone_str);
}

CHECK_OM_CALL(om_delete(telephones));

```

The **om_get()** call makes a copy of all or a selected set of parts of a private object. The copy is a service-generated public object that is accessible to the application program. The application program extracts the list of telephone numbers from this copy.

Required Input Parameters: The **om_get()** function requires the following input parameters:

- A private object
- A set of exclusions
- A set of OM attributes to be included in the copy
- A flag to indicate whether local string processing is required
- The position of the first value to be copied (the base value)
- The position within each OM attribute that is one beyond the last attribute to be included in the copy (indicating the scope of the copy)
- The public object that is a copy of the private object
- The number of OM attribute descriptors returned in the public object.

In the code fragment from **example.c**, the private object **result**, is input to **om_get()**.

The next parameter, the *exclusions* parameter, reduces the copy to a prescribed portion of the original. The exclusions apply to the OM attributes of the object, but not to those of subobjects. The possibilities for determining the combinations of types, values, subobjects, and descriptors to be excluded depend on the creativity of the programmer. For a detailed description of all the exclusion possibilities, see the *z/OS DCE Application Development Reference*. The values chosen for the **om_get()** calls in **example.c** are described below. These exclusion values are:

- **OM_EXCLUDE_ALL_BUT_THESE_TYPES**
- **OM_EXCLUDE_SUBOBJECTS**

Each value indicates an exclusion, as defined by **om_get()**, and is chosen from the set of exclusions; alternatively, the single value **OM_NO_EXCLUSIONS** can be chosen, which selects the entire object. Each value, except **OM_NO_EXCLUSIONS**, is represented by a distinct bit; the presence of the value is represented as 1, and its absence as 0. Multiple exclusions are requested by adding or ORing the values that indicate the individual exclusions.

OM_EXCLUDE_ALL_BUT_THESE_TYPES indicates that the OM attribute included are only the ones defined in the list of included types supplied in the next parameter, **entry_list**.

OM_EXCLUDE_SUBOBJECTS indicates that for each value whose syntax is **OM_S_OBJECT**, a descriptor containing an object handle for the original private subobject will be returned, rather than a public copy of it. This handle makes that subobject accessible for use in subsequent function calls. This exclusion provides a means to examine an object one level at a time. The object the handle points to is used in the next **om_get()** call to get the next level.

The *entry_list* parameter is declared in **example.c** as data type **OM_type** and initialized as a two-cell array with values **DS_ENTRY** and **NULL** terminator. **DS_ENTRY** specifies the single OM attribute type included for that **om_get()** call. The 0 (zero) marks the end of the OM attribute list.

The next parameter, **OM_TRUE**, tells the interface to return string values in the local EBCDIC code page.

The next two parameters set the initial and limiting value to 0 (zero), meaning that no specific values are to be excluded.

The final two parameters are output parameters: **entry**, a pointer to a service-generated public object deposited by **om_get()** in the workspace, and **total_num**, an integer. Both **entry** and **total_num** are available for examination by the application program.

Extracting the Data from the Read Result: The **entry** parameter contains the result of processing by **om_get()** of the **read** parameter generated by the **ds_read()** operation. A successful call to **ds_read()** returns an instance of OM class **DS_C_READ_RESULT** in the private object **result**. **DS_C_READ_RESULT** contains the information extracted from the directory entry of the target object.

The **om_get()** function call creates a public object to make the information contained in **result** available to the application program. The **entry** parameter is defined as data type **OM_public_object**. As such, it is composed of several nested layers of subobjects that contain entry information, OM attributes, and OM attribute values, as shown in Figure 43 on page 132. The series of **om_get()** calls removes these layers of objects to extract a list of telephone numbers.

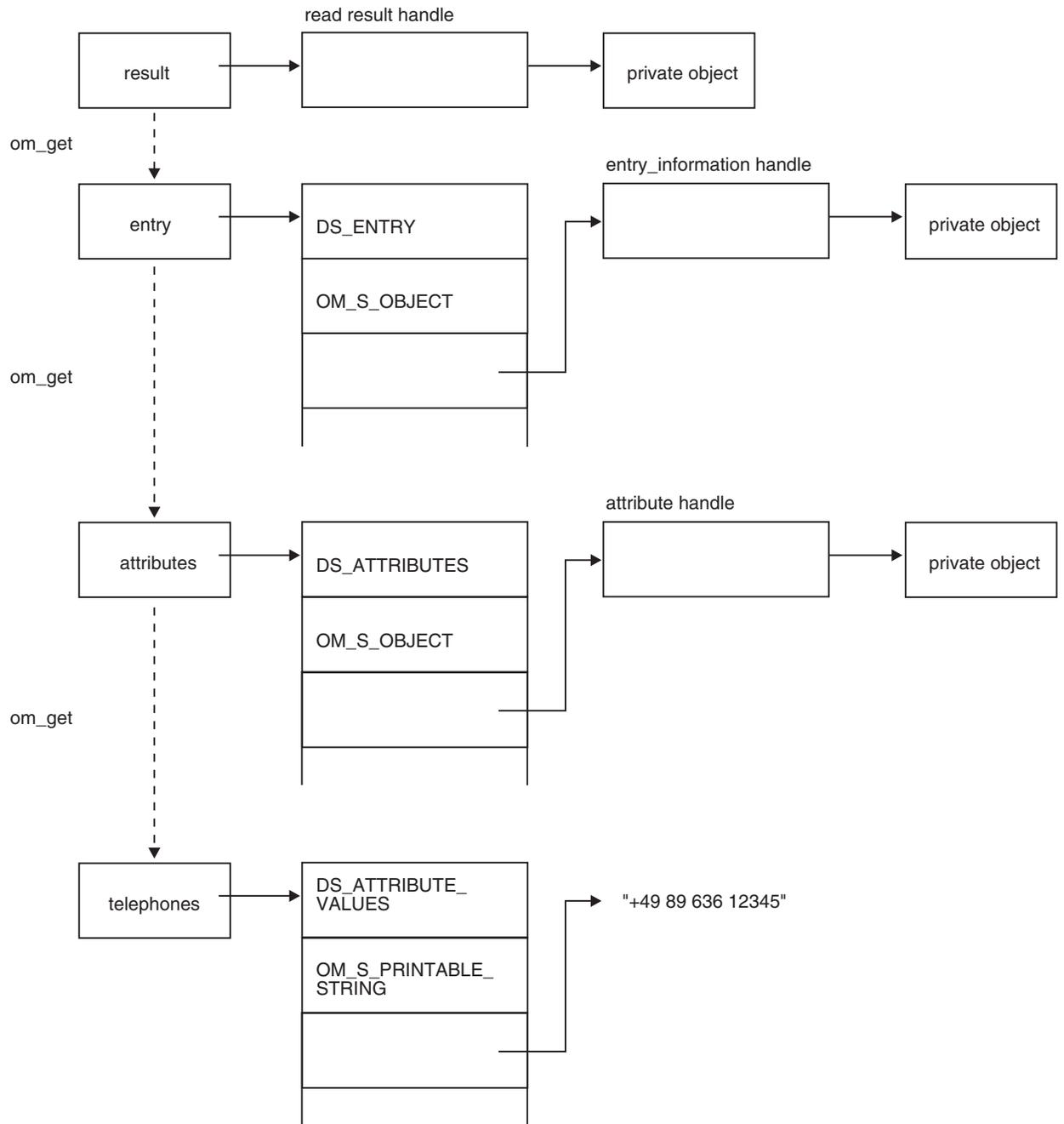


Figure 43. Extracting Information Using `om_get()`

Figure 43 also shows that the process of exposing the subobjects continues while the syntax of the subobjects is **OM_S_OBJECT**. In effect, `example.c` is reversing the process of building up a series of public objects as was necessary for providing input to `ds_read()`, namely the distinguished name of **Peter Piper** and the descriptor list for `entry_information_selection` (the **DS_C_ENTRY_INFO_SELECTION** OM object).

Return Codes: XOM API function calls return a value of type **OM_return_code** that indicates whether the function succeeded. If the function is successful, the value of the return code is set to **OM_SUCCESS**. If the function fails, it returns one of the values listed in Chapter 18, “XOM Service Interface” on page 331. The constants for **OM_return_code** are defined in the **xom.h** header file. In the **example.c** program, a C language macro named **CHECK_OM_CALL** is used to check the return code from each XOM API function call.

XOM API Header Files

The XOM API includes the header file **xom.h**. This header file is composed of declarations defining the C workspace interface. It supplies type definitions, symbolic constant definitions, and macro definitions.

XOM Type Definitions and Symbolic Constant Definitions

The **xom.h** header file includes **typedef** statements that define the data types of all OM objects used in the interface. It also provides definitions of symbolic constants used by the interface.

Refer to *z/OS DCE Application Development Reference* for a listing of the **xom.h** header file.

XOM API Macros

XOM API provides several macros that are useful in defining public objects in your application programs. These macros are defined in the **xom.h** header file:

- **OM_IMPORT**
Makes object identifier symbolic constants available within a C source module.
- **OM_EXPORT**
Allocates memory and initializes object identifier symbolic constants within a C source module.
- **OM_OID_DESC**
Initializes the type, syntax, and value of an OM attribute that holds an object identifier.
- **OM_NULL_DESCRIPTOR**
Marks the end of a client-generated public object.
- **OMP_LENGTH**
Calculates the length of an object identifier.
- **OM_STRING**
Creates a data value of a string data type.

The OM_EXPORT and OM_IMPORT Macros: You may find it convenient to export all the names used in your programs from the same C source module. **OM_EXPORT** allocates memory for the constants that represent an object OM class or an object identifier as shown in the following code fragment from **example.c**:

```

/* Define necessary Object Identifier constants
 */
OM_EXPORT(DS_A_COMMON_NAME)
OM_EXPORT(DS_A_COUNTRY_NAME)
OM_EXPORT(DS_A_ORG_NAME)
OM_EXPORT(DS_A_ORG_UNIT_NAME)
OM_EXPORT(DS_A_PHONE_NBR)
OM_EXPORT(DS_C_AVA)
OM_EXPORT(DS_C_DS_DN)
OM_EXPORT(DS_C_DS_RDN)
OM_EXPORT(DS_C_ENTRY_INFO_SELECTION)

```

In this code fragment, object identifier constants that represent OM classes that are defined in the **xds.h** and **xdsbdcp.h** header files are exported to the main program module. The object identifier constants are defined in **xds.h** with the **OMP_O_** prefix followed by the variable name for the object identifier. The constant itself provides the hexadecimal value of the object identifier string.

The **OM_EXPORT** macro takes the OM class name as input and creates two new data structures: a character string and a structure of type **OM_string**. These are normally used outside of any C function definitions, and so global variables are declared which represent the object identifiers. The structure of type **OM_string** contains a length and a pointer to a string that can be used later in an application program by the **OM_OID_DESC** macro to initialize the value of an object identifier.

OM_IMPORT marks the identifiers as external for the compiler. It is used if **OM_EXPORT** is called in a different file from where its values are referenced. (**OM_IMPORT** is not used in **example.c** because **OM_EXPORT** is called in the file where the object identifiers are referenced.) **OM_IMPORT** allows the global variables defined in one C file for object identifiers to be used in another C C file which is part of that same program.

The OM_OID_DESC Macro: The **OM_OID_DESC** macro initializes the type, syntax, and value of an OM attribute that holds an object identifier; in other words, it initializes **OM_descriptor**. It takes as input an OM attribute type and the name of an object identifier. The object identifier should have already been exported to the program module, as shown in the previous section.

The output of the macro is an **OM_descriptor** composed of a type, syntax, and value. The type is the name of the OM class. The syntax is **OM_S_OBJECT_IDENTIFIER**. The value is a two-member structure with the length of the object identifier and a pointer to the actual object identifier string.

OM_OID_DESC calls **OMP_LENGTH** to calculate the length of the object identifier string.

The following code fragment from **xom.h** shows the **OM_OID_DESC** and **OMP_LENGTH** macros:

```

/* Private macro to calculate length
 * of an object identifier
 */
#define OMP_LENGTH(oid_string) (sizeof(OMP_O_##oid_string)-1)

/* Macro to initialize the syntax and value
 * of an object identifier
 */
#define OM_OID_DESC(type, oid_name)
    { (type), OM_S_OBJECT_IDENTIFIER_STRING,
      { OMP_LENGTH(oid_name) , OMP_D_##oid_name } }

```

The OM_NULL_DESCRIPTOR Macro: The **OM_NULL_DESCRIPTOR** macro marks the end of a client-generated public object by setting the type to **OM_NO_MORE_TYPES**, the syntax to **OM_S_NO_MORE_SYNTAXES**, and value to zero length and a **NULL** string.

The OM_STRING Macro: The **OM_STRING** macro creates a string data value. Data strings are of type **OM_string**, as shown from this code fragment from the **xom.h** header file:

```
/* String */

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

#define OM_STRING(string) \
    { (OM_string_length)(sizeof(string)-1), string }
```

A string is specified in terms of its length or whether or not it terminates with a **NULL**. **OM_string_length** is the number of octets by which the string is represented, or it is the **OM_LENGTH_UNSPECIFIED** value if the string terminates with a **NULL**.

The bits of a bit string are represented as a sequence of octets. The first octet stores the number of unused bits in the last octet. The bits in the bit string, beginning with the first bit and proceeding to the trailing bit, are placed in bits 7 to 0 of the second octet. These are followed by bits 7 to 0 of the third octet, then by bits 7 to 0 of each octet in turn, followed by as many bits as are required of the final octet commencing with bit 7.

For example, a bit string representing the binary value '1001001111'B would have the form: \x06\x93\xC0. That is:

```

    x'06'      x'93'      x'C0'

                xxxxxxxx-----xx-----> original bit string
00000110     10010011     11000000
>>>>----->-----xxxxxx----> 6 unused bits
```

This describes the basic encoding rule for bit strings. For a complete description of the basic encoding rules, see ISO 8825.

Chapter 6. XDS Programming

XDS API defines an application programming interface to directory services in the X/Open Common Applications Environment as defined in the *X/Open Portability Guide*. This interface is based on the 1988 CCITT X.500 Series of Recommendations and the ISO 9594 Standard. This joint standard is referred to from this point on simply as X.500.

This chapter describes the purpose and function of XDS API functions in a general way. Refer to the *z/OS DCE Application Development Reference* for complete and detailed reference information on specific function calls.

Subsequent sections describe the following types of XDS functions:

- XDS Interface Management Functions, which interact with the XDS interface
- Directory Connection Management Functions, which initiate, manage, and terminate connections with the directory
- Directory Operation Functions, which perform operations on a directory.

Note:

The DCE XDS API does not support asynchronous operations from within the same thread. If an application requires asynchronous XDS operations, then it should use multiple threads to achieve this functionality. Please refer to Chapter 8, “Using Threads with the XDS/XOM API” on page 191 for information on using the XDS/XOM API in a multithreaded application.

Because of this, the **ds_search()** and **ds_modify_rdn()** routines are not supported. These routines return **DS_E_UNAVAILABLE** if a GDS name is used in the call. A **DS_E_UNWILLING_TO_PERFORM** is returned if a CDS name is used. A **ds_abandon()** call returns a **DS_C_ABANDON_FAILED (DS_E_TOO_LATE)** error. A **ds_receive_result()** call returns with **DS_Status** set to **DS_SUCCESS**, and the *completion_flag_return* parameter set to **DS_NO_OUTSTANDING_OPERATION**.

Following are the names of the complete XDS example programs that you can find in Chapter 7, “Example Application Programs” on page 159:

- **example.c (example.h)**
- **teldir.c**

XDS Interface Management Functions

XDS API defines a set of functions that only interact with the XDS interface and have no counterpart in the Directory standard definition:

- **ds_initialize()**
- **ds_version()**
- **ds_shutdown()**.

These interface functions perform operations that involve the initialization, management, and termination of sessions with the XDS interface service.

The `ds_initialize()` Function Call

Every application program must first call `ds_initialize()` to establish a workspace where objects returned by the Directory Service are deposited. The `ds_initialize()` function must be called before any other directory interface functions are called.

The `ds_initialize()` call returns a handle (or pointer) to a workspace. The application program performs operations on OM objects in this workspace. OM objects created in this workspace can be used as input parameters to the other directory interface functions. In addition, objects returned by the Directory Service are deposited in the workspace.

Within the following code fragment from `example.c`, a workspace is initialized (the declaration of the variable `workspace` and the call to `ds_initialize()` are found in different sections of the program):

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

    /*
     * Perform the Directory operations:
     * (1) Initialize the Directory Service and get an OM workspace.
     * (2) bind a default directory session.
     * (3) read the telephone number of "name".
     * (4) terminate the directory session.
     */

    CHECK_DS_CALL((OM_object) !(workspace=ds_initialize()));
}
```

`OM_workspace` is a type definition in the `xom.h` header file defined as a pointer to `void`. A void pointer is a generic pointer that can point to any data type. The variable `workspace` is declared as data type `OM_workspace`. The return value is assigned to the variable `workspace` and the `CHECK_DS_CALL` macro determines if the call is successful. `CHECK_DS_CALL` is an error handling macro that is defined in `example.h`.

The `ds_initialize()` call returns a handle to a workspace in which OM objects can be created and manipulated. The `ds_initialize()` call returns `NULL` if it fails.

The `ds_version()` Function Call

The `ds_version()` call negotiates features of the directory interface. These features are collected into packages that define the scope of the service. Packages define such things as object identifiers for Directory and OM classes and OM attributes, enumerated types, structures, and OM object constants.

XDS API defines the following packages in separate header files as part of the XDS API software product:

- Directory Service Package
The Directory Service Package contains the OM classes and OM attributes used to interact with the Directory Service. This package is contained in the `xds.h` header file.
- Basic Directory Contents Package

The Basic Directory Contents Package contains OM classes and OM attributes that represent values of selected attributes and selected objects defined in the X.500 standard. This package is contained in the **xdsbdcp.h** header file.

- Strong Authentication Package

The Strong Authentication Package contains OM classes and OM attributes that represent values of security attributes and objects defined in the X.500 standard. This package is contained in the **xdssap.h** header file.

- Global Directory Service Extension Package

The Global Directory Service Extension Package contains the OM classes and OM attributes that are required for GDS. This package is contained in the **xdsfds.h** header file.

- MHS Directory User Package

The MHS (Message Handling Systems) Directory User Package contains the OM classes and OM attributes that are required for Electronic Mail API. This package is contained in the **xdsmdup.h** header file.

The application program, the client, uses **ds_version()** to *negotiate* the scope of the services the Directory Service will provide to the client. A **ds_version()** function call includes a list of features (or packages) that the client wants to include as part of the interface. The features are object identifiers that represent packages supported by the DCE XDS API. The service returns a list of boolean values to indicate if the package was successfully negotiated.

These features are assigned to the workspace that an application program initialized (as described in “The ds_initialize() Function Call” on page 138). In addition, an application program must include the header files for the appropriate packages as part of the source code.

It is not necessary to negotiate the Directory Service Package. It is a mandatory requirement for XDS API and as such it is included by default. The other packages listed previously are optional and require negotiation using **ds_version()**.

The following code fragment from **teldir.c** shows how an application builds up an array of object identifiers for the optional package to be negotiated: the Basic Directory Contents Package.

```
/*
 * To identify which packages we need for this program. We only need
 * the basic package because we are not doing anything fancy with
 * session parameters, etc.
 */
DS_feature featureList[] = {
    { OM_STRING(OMP_O_DS_BASIC_DIR_CONTENTS_PKG), OM_TRUE },
    { 0 }
};
```

The **OM_STRING** macro is provided for creating a data value of data type **OM_string** for octet strings and characters. The array of object identifiers is stored in *featureList*, the input parameter to **ds_version()**, as shown in the following code fragment from **teldir.c**:

```
/* STEP 3
 *
 * Pull in the packages that contain the XDS features we need.
 */
dsStatus = ds_version( featureList, xdsWorkspace );
if( dsStatus != DS_SUCCESS )
    handledSError( "ds_version()", dsStatus );
```

The `ds_shutdown()` Function Call

The `ds_shutdown()` call deletes the workspace established by `ds_initialize()` and enables the Directory Service to release resources. No other directory functions that reference that workspace can be called after this function.

The following code fragment from `teldir.c` demonstrates how the application closes the directory workspace by performing a `ds_shutdown()`.

```
/* STEP 9 */
dsStatus = ds_shutdown( xdsWorkspace );
if( dsStatus != DS_SUCCESS )
    handledSError( "ds_shutdown()", dsStatus );
```

Directory Connection Management Functions

The following subsections describe the XDS functions that initiate, manage, and terminate connections with the Directory.

A Directory Session

A directory session identifies the DSA to which a directory operation is sent. It also defines the characteristics of a session, such as the distinguished name of the requestor.

An application program can request a session with specific OM attributes tailored for the program's requirements. The application passes an instance of OM class `DC_C_SESSION` with the appropriate OM attributes, or it uses default parameters by passing the constant `DS_DEFAULT_SESSION` as a parameter to the `ds_bind()` function call.

The `ds_bind()` Function Call

The `ds_bind()` call establishes a session with the Directory. The `ds_bind()` call corresponds to the `DirectoryBind` function in the Abstract Service defined in the X.500 standard.

When a `ds_bind()` call is completed successfully, the directory returns a pointer to an OM private object of OM class `DC_C_SESSION`. This parameter is then passed as the first parameter to most interface function calls until a `ds_unbind()` is called to terminate the directory session.

XDS API supports multiple concurrent sessions so that an application can interact with the Directory using several identities and interact directly and concurrently with different parts of the Directory Service.

The following code fragment from `example.c` shows how an application binds to the GDS server (without credentials) using the default session:

```
CHECK_DS_CALL(ds_bind(DS_DEFAULT_SESSION, workspace, &session));
```

The `ds_unbind()` Function Call

The `ds_unbind()` call terminates a directory session and makes the session parameter unavailable for use with other interface functions. However, the unbound session can be modified by OM functions and used again as a parameter to `ds_bind()`. When the session parameter is no longer needed, it should be deleted using OM functions (such as `om_delete()`).

The following code fragment from `example.c` shows how the application closes the connection to the GDS server using `ds_unbind()`:

```
/* Close the connection to the GDS server.          */
if (ds_unbind(bound_session) != DS_SUCCESS)
    printf("ds_unbind() error\n");
```

The `ds_unbind()` call corresponds to the `DirectoryUnbind` function in the Abstract Service defined in the X.500 standard.

Automatic Connection Management

The XDS implementation does not support automatic connection management. A DSA connection is established when `ds_bind()` is called and released when `ds_unbind()` is called.

XDS Interface Class Definitions

The XDS Interface Class Definitions are described in detail in Chapter 11, “XDS Class Definitions” on page 241. The OM attribute types, syntax, and values and inheritance properties are described for each OM class.

A good way to begin to understand how the OM class hierarchy is structured, and the relationship between OM classes and OM attributes to the service provided by the Directory Service Package, is to look up one of the OM classes listed in Chapter 11, “XDS Class Definitions” on page 241.

For example, `DS_C_FILTER` inherits the OM attributes from its superclass `OM_C_OBJECT`, as do all OM classes. `OM_C_OBJECT`, as defined in Chapter 5, “XOM Programming” on page 97, has one OM attribute, `OM_CLASS`, which has the value of an object identifier string that identifies the numeric representation of the object's OM class. `DS_C_FILTER`, on the other hand, has several OM attributes.

The purpose of `DS_C_FILTER` is to select or reject an object on the basis of information in its directory entry. It has the following OM attributes:

- `DS_FILTER_ITEMS`
- `DS_FILTERS`
- `DS_FILTER_TYPE`.

Two of these OM attributes, `DS_FILTER_ITEMS` and `DS_FILTERS`, have values that are OM object classes themselves. The value of the OM attribute `DS_FILTER_ITEMS` is `DS_C_FILTER_ITEM`, which is an OM class. `DS_C_FILTER_ITEM` is a component of a filter and defines the nature of the filter. The value of the OM attribute `DS_FILTERS` is `DS_C_FILTER`, an OM class. Thus, `DS_FILTERS` defines a collection of filters. The OM attribute `DS_FILTER_TYPE` has a value that is an enumerated type, which takes one of the values: `DS_AND`, `DS_OR`, or `DS_NOT`.

The DS_C_CONTEXT Parameter

The OM class **DS_C_CONTEXT** is the second parameter to every Directory Service request. **DS_C_CONTEXT** defines the characteristics of the Directory Service interaction that are specific to a particular Directory Service operation. These characteristics are divided into three categories of OM attributes: common parameters, service controls, and local controls. The XDS interface defines **DS_DEFAULT_CONTEXT** to simplify usage of this parameter.

Common parameters affect the processing of each Directory Service operation.

Service controls indicate how the Directory Service should handle requests. Included in this category are decisions about whether or not chaining is permitted, the priority of requests, the scope of referral, and the maximum number of objects about which a function should return information.

Local controls include asynchronous support and automatic continuation; XDS does not support asynchronous operations from within the same thread. Applications requiring asynchronous use of the XDS/XOM API should use threads as defined in Chapter 8, "Using Threads with the XDS/XOM API" on page 191.

Directory Class Definitions

The X.500 standards define a number of attribute types and classes. These definitions allow the creation and maintenance of directory entries for a number of common objects so that the representation of all such objects is the same throughout the directory. The Basic Directory Contents Package contains OM classes and OM attributes that model the X.500 attribute types and classes.

The X.500 object classes and attributes are defined in the following documents published by CCITT. These are the objects and the associated attributes that will be the targets of Directory Service operations in your application programs.

- *The Directory: Selected Attribute Types*
- *The Directory: Selected Object Classes*

Table 52 on page 277 describes the OM classes and OM attributes and their object identifiers that model these X.500 objects and attributes in detail.

The table contains similar categories of information as other attribute tables defined in the **Directory Service Package**. These information categories include the following:

- OM Value syntax
- Value Length
- Multivalued
- Matching Rules.

The OM Value Syntax describes the structure of the values of an OM attribute. The Value Length gives the range of lengths permitted for the string types. Table 52 on page 277 also indicates if the attribute can have multiple values.

The CCITT standards define matching rules that are used for determining whether two values are equal, for ordering two values, or for identifying one value as a substring of another in Directory Service operations.

The GDS administrator maintains the Directory Service and determines the structure of the DIT as defined by the GDS schema. The GDS standard (or default) schema is based on the recommendations in the CCITT documents mentioned previously.

Recall that the Structure Rule Table (SRT) of the GDS schema defines the structure of the DIT, the Object Class Table (OCT) defines class inheritance properties, and the Attribute Table (AT) defines the mandatory and optional attributes for each class. You will find it useful to familiarize yourself with the existing schema when developing an application program that will access the directory. The public objects that your programs will create (using OM classes and OM attributes) are modeled after objects and attributes in the Directory.

Directory Operation Functions

The X.500 standard defines the operations provided by the directory in a document called the *Abstract Service Definition*. DCE implements this standard with XDS API function calls. The XDS API functions allow an application program to interact with the Directory Service. The standard divides these interactions into three general categories: read, search, and modify.

The XDS API functions correspond to the Abstract Service functions defined in the X.500 standard, as shown in Table 19.

Table 19. Mapping of XDS API Functions to the Abstract Services

XDS Function Call	Abstract Service Equivalent
ds_read()	Read
ds_compare()	Compare
ds_list()	List
ds_search()	Search
ds_add_entry()	AddEntry
ds_remove_entry()	RemoveEntry
ds_modify_entry()	ModifyEntry
ds_modify_rdn()	ModifyRDN

Note: **ds_search()** and **ds_modify_rdn()** are not supported. These routines return **DS_E_UNAVAILABLE** if a GDS name is used in the call, and **DS_E_UNWILLING_TO_PERFORM** if a CDS name is used.

Directory Read Operations

Read functions retrieve information from specific named entries in the Directory where names are mapped to attributes. This is analogous to looking up some information about a name in the *White Pages* phone directory.

XDS API implements the following read functions:

- **ds_read()**

The requester supplies a distinguished name and selection criteria that specify what information from the entry is requested. The values of requested attributes or just the attribute types are returned by the DSA.

- **ds_compare()**

The requester gives a distinguished name and an Attribute Value Assertion (AVA). If the AVA is true for the named entry, a value of TRUE is returned by the DSA.

For example, a typical read operation could request the telephone number of a particular employee. A read request would submit the distinguished name of the employee with an indication to return its telephone number: **/C=us/O=sni/OU=sales/CN=John Smith**.

Reading an Entry from the Directory

The following sections describe a typical read operation using the **ds_read()** function call. It includes a description of tasks directly related to the read operation. They do not include service-related tasks such as initializing the interface allocating an OM workspace, and binding to the directory. These tasks were described earlier in this chapter. The following sections also do not describe the process of extracting information from the workspace using XOM functions. Refer to Chapter 5, “XOM Programming” on page 97 for a description of how to use XOM functions to access the workspace.

A typical read operation involves the following steps:

1. Define the necessary object identifier constants for the OM classes and OM attributes that will define public objects for input to **ds_read()** using the **OM_EXPORT** macro.
2. Declare the variables that will contain the output from the XDS functions to be used in the application.
3. Build public objects (descriptor lists) for the *name* parameter to **ds_read()**.
4. Create a descriptor list for the *selection* parameter to **ds_read()** that selects the type and scope of information in your request.
5. Perform the read operation.

These steps are demonstrated in the following code fragments from **example.c**. (Refer to Chapter 7, “Example Application Programs” on page 159 for a complete program listing.) The program reads the telephone numbers of a given target entry.

Step 1: Export Object Identifiers for Required Directory Classes and Attributes

In the following code fragment from **example.c**, the **OM_EXPORT** macro allocates memory for the constants that represent the OM object identifiers for object classes and directory attributes required for the read operation:

```
/* Define necessary Object Identifier constants
*/
OM_EXPORT(DSX_TYPELESS_RDN)
OM_EXPORT(DS_A_COUNTRY_NAME)
OM_EXPORT(DS_A_ORG_NAME)
OM_EXPORT(DS_A_ORG_UNIT_NAME)
OM_EXPORT(DS_A_PHONE_NBR)
OM_EXPORT(DS_C_AVA)
OM_EXPORT(DS_C_DS_DN)
OM_EXPORT(DS_C_DS_RDN)
OM_EXPORT(DS_C_ENTRY_INFO_SELECTION)
```

The **OM_EXPORT** macro performs the following steps:

1. It defines a character array called **OMP_D_class_name** input argument. where *class_name* is the input argument.
2. It initializes this array to the value of a character string called **OMP_O_class_name** where *class_name* is the input parameter. This value has already been defined in a header file.
3. It defines an **OM_string** data structure with name *class_name*.
4. It initializes the **OM_string** data structure's first component to the length of the array initialized in Step 2 above and initializes the second component to a pointer to the value of the array initialized in Step 2 above.

The net result is that a static **OM_string** structure is declared with name *class_name*. This structure is quite useful when building OM objects.

Step 2: Declare Local Variables

The local variables *session*, *result*, and *invoke_id* are defined in the following code fragment from **example.c**:

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

These data types are defined by **typedef** statements in the **xom.h** header file. The *session* and *result* variables are defined as data type **OM_private_object** because they are returned by **ds_bind()** and **ds_read()** as private objects. Because asynchronous operations are not supported, the *invoke_id* parameter is ignored. The *invoke_id* parameter must be supplied to the XDS functions as described in the *z/OS DCE Application Development Reference* but its return value should be ignored.

Values in *error* and *return_code* are returned by XOM and XDS functions to indicate whether a call was successful. The *workspace* variable is of type **OM_workspace** and is used when establishing an OM workspace. The *total_num* variable is of type **OM_value_position** to indicate the number of attribute descriptors returned in the public object by the function **om_get()** based on the inclusion and exclusion parameters specified.

Step 3: Build Public Objects

A **ds_read()** function call can take a public object as an input parameter. A public object is generated by an application program and contains the information required to access a target directory object. This information includes the AVAs and RDNs that make up a distinguished name of an entry in the directory.

A public object is created using OM classes and OM attributes. These OM classes and OM attributes model the target object entry in the directory and provide other information required by the Directory Service to access the directory. In this case, the target object entry in the Directory is the entry for **Peter Piper**.

“Public Objects” on page 102 describes how to create the required public objects for the **ds_read()** function call using macros and data structures defined in the XDS and XOM API header files.

The purpose of building the public objects for AVAs and RDNs is to provide the public object that represents a distinguished name. The distinguished name public object is stored in the array of descriptors called *name* and provided as an input parameter to the **ds_read()** function call.

Step 4: Create an Entry-Information-Selection Parameter

The distinguished name for **Peter Piper** is an entry in the directory that the application is designed to access. The *selection* parameter of the **ds_read()** operation function call tailors its results to obtain just part of the required entry. Information on all attributes, no attributes, or a specific group of attributes can be chosen. Attribute types are always returned, but the attribute values need not be returned.

The value of the parameter is a public object (descriptor list) that is an instance of OM class **DS_C_ENTRY_INFO_SELECTION** as shown in the following code fragment from **example.c**:

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

DS_C_ENTRY_INFO_SELECTION is a subclass of **OM_C_OBJECT**. (This information is supplied in the description of this class in Chapter 11, "XDS Class Definitions" on page 241.) As such, **DS_C_ENTRY_INFO_SELECTION** inherits the OM attributes of **OM_C_OBJECT**. The only OM attribute of **OM_C_OBJECT** is **OM_CLASS**. **OM_CLASS** identifies an object's class, which in this case is **DS_C_ENTRY_INFO_SELECTION**. **DS_C_ENTRY_INFO_SELECTION** identifies information to be extracted from a directory entry and has the following OM attributes:

- **OM_CLASS** (inherited from **OM_C_OBJECT**)
- **DS_ALL_ATTRIBUTES**
- **DS_ATTRIBUTES_SELECTED**
- **DS_INFO_TYPE**.

As part of a **ds_read()** or **ds_search()** function call, **DS_ALL_ATTRIBUTES** specifies to the Directory Service which attributes of a directory entry are relevant to the application program. It can take the values **OM_TRUE** or **OM_FALSE** because the **DS_ALL_ATTRIBUTES** descriptor is defined to be of syntax **OM_S_BOOLEAN**. The value **OM_TRUE** indicates that information is requested on all attributes in the directory entry. The value **OM_FALSE** used in the preceding sample code fragment indicates that information is only requested on those attributes that are listed in the OM attribute **DS_ATTRIBUTES_SELECTED**.

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

OM_S_OBJECT_IDENTIFIER_STRING contains an octet string of BER-encoded integers. The integers are decimal representations of object identifiers. The object identifiers represent attribute types that are requested from the directory entry. In the preceding code fragment above, the string value is the attribute

name **DS_A_PHONE_NBR** because the purpose of the read call is to read a list of telephone numbers from the directory.

DS_INFO_TYPE identifies what information is to be extracted from each attribute identified. The syntax of the value is specified as Enum(**DS_Information_Type**). **DS_INFO_TYPE** is an enumerated type that has two possible values: **DS_TYPES_ONLY** and **DS_TYPES_AND_VALUES**. **DS_TYPES_ONLY** indicates that only the attribute types of the selected attributes in the entry are returned by the Directory Service operation. **DS_TYPES_AND_VALUES** indicates that both the attribute types and the attribute values of the selected attributes in the entry are returned. The code fragment from **example.c** shown previously defines the value of **DS_INFO_TYPE** as **DS_TYPES_AND_VALUES** because the program wants to get the actual telephone numbers.

Step 5: Perform the Read Operation

The following code fragment from **example.c** shows the **ds_read()** function call and the XDS calls that precede it:

```
/*
 * Perform the Directory operations:
 * (1) Initialize the Directory Service
 * and get an OM workspace.
 * (2) bind a default directory session.
 * (3) read the telephone number of "name".
 * (4) terminate the directory session.
 */

CHECK_DS_CALL((OM_object) !(workspace = ds_initialize()));

CHECK_DS_CALL(ds_version(bdcp_package, workspace));

CHECK_DS_CALL(ds_bind(DS_DEFAULT_SESSION, workspace,
                    &session));

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

CHECK_DS_CALL is an error-checking macro defined in the **example.h** header file that is included by **example.c**. The **ds_read()** call returns a return code of type **DS_status** to indicate whether or not the read operation was completed successfully. If the call was successful, **ds_read()** returns the value **DS_SUCCESS**. If the call fails, it returns an error code. (Refer to “XDS Errors” on page 241 for a comprehensive list of error codes.) **CHECK_DS_CALL** interprets this return value and returns successfully to the program, or branches to an error-handling routine.

The *session* input parameter is a private object generated by **ds_bind()** prior to the **ds_read()** call as shown in the preceding code fragment. The *context* input parameter describes the characteristics of a Directory interaction. Most XDS API function calls require these two input parameters because they define the operating parameters of a session with a GDS server. (Sessions are described in “A Directory Session” on page 140; contexts are described in “The DS_C_CONTEXT Parameter” on page 142.)

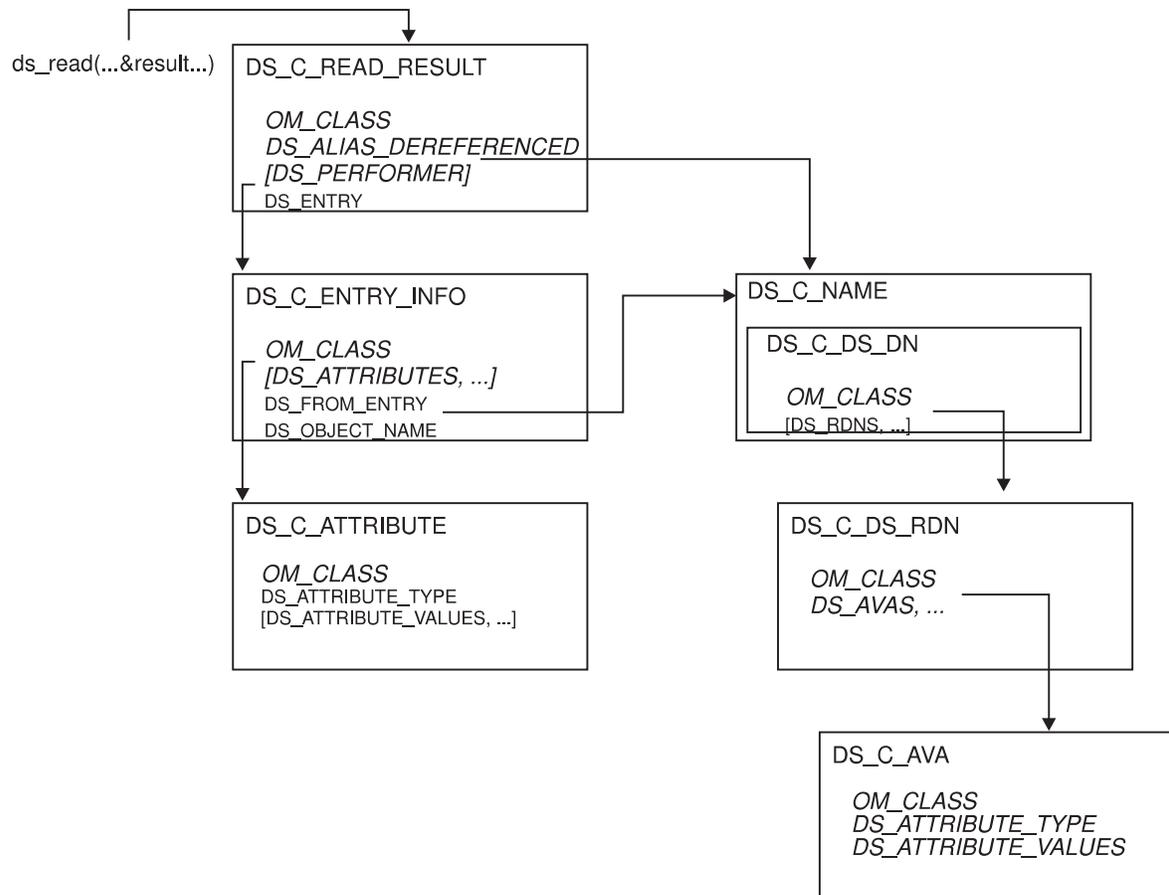
The result of a Directory Service request is returned in a private object (in this case *result*) that is appropriate to the type of operation. The components of the result are represented by OM attributes in the operations result object:

- **DS_C_COMPARE_RESULT**
Returned by **ds_compare()**

- **DS_C_LIST_RESULT**
Returned by **ds_list()**
- **DS_C_READ_RESULT**
Returned by **ds_read()**
- **DS_C_SEARCH_RESULT**
Returned by **ds_search()**.

The OM class returned by **ds_read()** is **DS_C_READ_RESULT**. The OM class returned by the **ds_compare()** call is **DS_C_COMPARE_RESULT** and so on. (Refer to *z/OS DCE Application Development Reference* for a description of the OM classes associated with a particular function call; refer to Chapter 11, “XDS Class Definitions” on page 241 for a full description of the OM attributes, syntaxes, and values associated with these OM classes.)

The super-classes, subclasses, and OM attributes for **DS_C_READ_RESULT** are shown in Figure 44 on page 149.



KEY:

- └─┬─┘ points to subobjects
- BOLD** OM class
- BOLD ITALICS** abstract OM class
- ITALICS* inherited OM attribute
- [] optional OM attribute
- , ... multi-valued OM attribute

Figure 44. Output from `ds_read()`: `DS_C_READ_RESULT`

The *result* value is returned to the workspace in private implementation-specific format. As such, it cannot be read directly by an application program, but requires a series of `om_get()` function calls to extract the requested information from it. The following code fragment from `example.c` shows how a series of `om_get()` calls extracts the list of telephone numbers associated with the distinguished name for **Peter Piper**. “Using the OM Function Calls” on page 129 describes this extraction process in detail.

```

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

```

```

CHECK_OM_CALL( om_get()(result,

```

```

    OM_EXCLUDE_ALL_BUT_THESE_TYPES + OM_EXCLUDE_SUBOBJECTS,
    entry_list, OM_TRUE, 0, 0, &entry, &total_num));

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

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

```

Directory Search Operations

Search functions can be used to browse through the Directory Information Tree (DIT). For example, a search request could supply the distinguished name of an entry and request a list of the distinguished names of the children of that entry that meets certain criteria.

XDS API implements the following search operations:

- **ds_list**

The requestor supplies a distinguished name. The Directory Service returns a list of the immediate subordinates of the named entry.

- **ds_search()**

The requestor supplies search criteria known as a *filter*. The user names a subtree of the DIT, specifies some target attribute types, and formulates an expression combining a number of attributes using logical AND, OR, or NOT operators. The Directory Service returns information from all of the entries within the specified portion of the DIT that match the filter.

Note: **ds_search()** is not supported.

Directory Modify Operations

Modify functions alter information in the directory. For example, if an employee of an Organizational Unit transfers to a new Organizational Unit, a typical modify request would modify the **OU** name attribute in the person's directory entry to reflect the change.

XDS API implements the following modify functions:

- **ds_modify_entry()**

The requestor gives a distinguished name and a list of modifications to the named entry. The Directory Service carries out the specified changes if the user requesting the change has the required access rights.

- **ds_add_entry()**

The requestor gives a distinguished name and values for a new entry. The entry is added as a leaf node in the DIT if the user requesting the change has the required access rights.

- **ds_remove_entry()**

The requestor gives a distinguished name. The entry with that name is removed if the user requesting the change has the required access rights.

- **ds_modify_rdn()**

The requestor gives a distinguished name and a new Relative Distinguished Name (RDN) for the entry. The directory changes the entry's RDN if the user requesting the change has the required access rights.

Note: `ds_modify_rdn()` is not supported.

Note that `ds_add_entry()`, `ds_remove_entry()`, and `ds_modify_rdn()` only apply to leaf entries. They are not intended to provide a general facility for building and manipulating the DIT.

Modifying Directory Entries

This section describes a modification and subsequent listing of the DIT using the `ds_add_entry()`, `ds_list()`, and `ds_remove_entry()` function calls. It includes a description of tasks directly related to these operations and does not include service-related tasks. It does not include a `ds_modify_entry()` function call. The modify operation is used in the context of the X.500 *Abstract Service Definition*.

A typical operation to add, remove, or list an entry involves following the same basic steps that were defined previously for read and search operations:

1. Define the necessary object identifier constants for the OM classes and OM attributes that will define public objects for input to the function calls by using the **OM_EXPORT** macro.
2. Declare the variables that will contain the output from the XDS functions you will use in your application.
3. Build public objects (descriptor lists) for the *name* parameters to the function calls.
4. Create descriptor lists for the attributes to be added, removed, or listed.
5. Perform the operations.

These steps are demonstrated in the following code fragments. The program adds two entries to the directory, then a list operation is performed on their superior entry, and finally the two entries are removed from the directory. The directory tree shown in Figure 45 is used in the program.

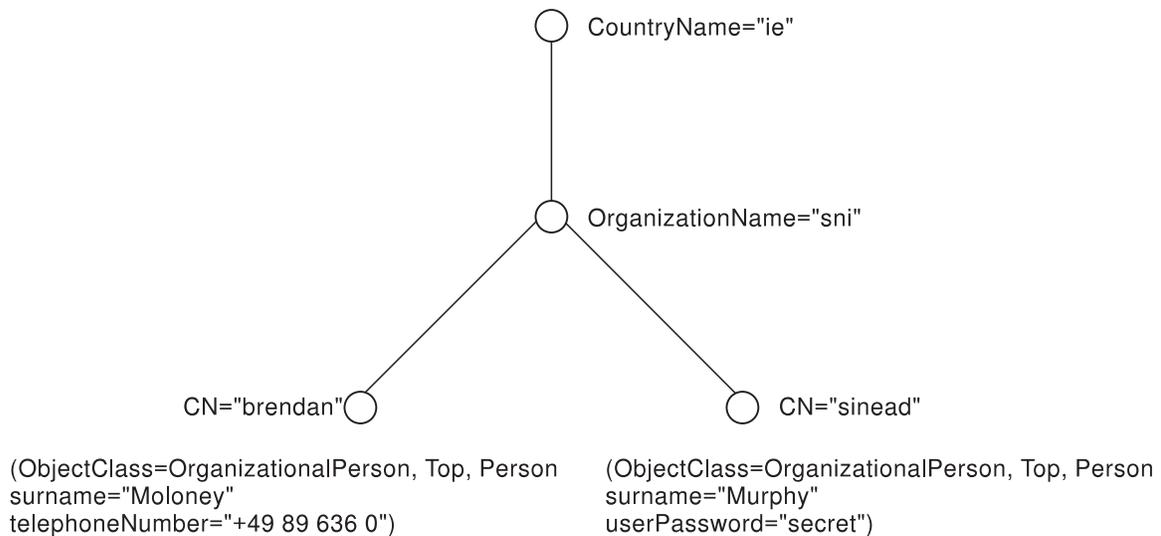


Figure 45. Sample Directory Tree

Step 1: Export Object Identifiers for Required Directory Classes and Attributes

In the following code fragment, the **OM_EXPORT** macro declares global constants that represent the object classes and attributes required for the add, list, and remove operations:

```
/* The application has to export the object identifiers */
/* it requires. */

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

OM_EXPORT (DS_A_COUNTRY_NAME)
OM_EXPORT (DS_A_ORG_NAME)
OM_EXPORT (DS_A_COMMON_NAME)
OM_EXPORT (DS_A_OBJECT_CLASS)
OM_EXPORT (DS_A_PHONE_NBR)
OM_EXPORT (DS_A_USER_PASSWORD)
OM_EXPORT (DS_A_SURNAME)

OM_EXPORT (DS_O_TOP)
OM_EXPORT (DS_O_PERSON)
OM_EXPORT (DS_O_ORG_PERSON)
```

Step 2: Declare Local Variables

The local variables *bound_session*, *result*, and *invoke_id* are defined in the following sample code fragment:

```
OM_private_object bound_session; /* Holds the Session object */
                                /* which is returned by */
                                /* ds_bind(). */
OM_private_object result;      /* Holds the list result */
                                /* object. */
OM_sint          invoke_id;    /* Integer for the invoke id */
                                /* returned by ds_search(). */
                                /* This parameter must be */
                                /* present even though it is */
                                /* ignored. */
```

These data types are defined in **typedef** statements in the **xom.h** header file. The *bound_session* and *result* variables are defined as data type **OM_private_object** because they are returned by **ds_bind()** and **ds_list()** operations to the workspace as private objects. Because asynchronous operations are not supported, the *invoke_id* parameter functionality is redundant. The *invoke_id* parameter must be supplied to the XDS functions as described in the *z/OS DCE Application Development Reference*, but its return value should be ignored.

Step 3: Build Public Objects

The public objects required by the `ds_add_entry()`, `ds_list()`, and `ds_remove_entry()` operations are defined in the following code fragment:

```
/* Build up descriptor lists for the following distinguished names: */
/* C=ie/O=sni */
/* C=ie/O=sni/CN=brendan */
/* C=ie/O=sni/CN=sinead */

static OM_descriptor ava_ie[] = {
    OM_OID_DESC(OM_CLASS, DS_C_AVA),
    OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_COUNTRY_NAME),
    {DS_ATTRIBUTE_VALUES, OM_S_PRINTABLE_STRING, OM_STRING("ie")},
    OM_NULL_DESCRIPTOR
};
static OM_descriptor ava_sni[] = {
    OM_OID_DESC(OM_CLASS, DS_C_AVA),
    OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_ORG_NAME),
    {DS_ATTRIBUTE_VALUES, OM_S_TELETEX_STRING, OM_STRING("sni")},
    OM_NULL_DESCRIPTOR
};
static OM_descriptor ava_brendan[] = {
    OM_OID_DESC(OM_CLASS, DS_C_AVA),
    OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_COMMON_NAME),
    {DS_ATTRIBUTE_VALUES, OM_S_TELETEX_STRING, OM_STRING("brendan")},
    OM_NULL_DESCRIPTOR
};
static OM_descriptor ava_sinead[] = {
    OM_OID_DESC(OM_CLASS, DS_C_AVA),
    OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_COMMON_NAME),
    {DS_ATTRIBUTE_VALUES, OM_S_TELETEX_STRING, OM_STRING("sinead")},
    OM_NULL_DESCRIPTOR
};
static OM_descriptor rdn_ie[] = {
    OM_OID_DESC(OM_CLASS, DS_C_DS_RDN),
    {DS_AVAS, OM_S_OBJECT, {0, ava_ie}},
    OM_NULL_DESCRIPTOR
};
static OM_descriptor rdn_sni[] = {
    OM_OID_DESC(OM_CLASS, DS_C_DS_RDN),
    {DS_AVAS, OM_S_OBJECT, {0, ava_sni}},
    OM_NULL_DESCRIPTOR
};
static OM_descriptor rdn_brendan[] = {
    OM_OID_DESC(OM_CLASS, DS_C_DS_RDN),
    {DS_AVAS, OM_S_OBJECT, {0, ava_brendan}},
    OM_NULL_DESCRIPTOR
};
static OM_descriptor rdn_sinead[] = {
    OM_OID_DESC(OM_CLASS, DS_C_DS_RDN),
    {DS_AVAS, OM_S_OBJECT, {0, ava_sinead}},
    OM_NULL_DESCRIPTOR
};
static OM_descriptor dn_sni[] = {
    OM_OID_DESC(OM_CLASS, DS_C_DS_DN),
    {DS_RDNS, OM_S_OBJECT, {0, rdn_ie}},
```

```

        {DS_RDNS,OM_S_OBJECT,{0,rdn_sni}},
        OM_NULL_DESCRIPTOR
};
static OM_descriptor dn_brendan[] = {
    OM_OID_DESC(OM_CLASS,DS_C_DS_DN),
    {DS_RDNS,OM_S_OBJECT,{0,rdn_ie}},
    {DS_RDNS,OM_S_OBJECT,{0,rdn_sni}},
    {DS_RDNS,OM_S_OBJECT,{0,rdn_brendan}},
    OM_NULL_DESCRIPTOR
};
static OM_descriptor dn_sinead[] = {
    OM_OID_DESC(OM_CLASS,DS_C_DS_DN),
    {DS_RDNS,OM_S_OBJECT,{0,rdn_ie}},
    {DS_RDNS,OM_S_OBJECT,{0,rdn_sni}},
    {DS_RDNS,OM_S_OBJECT,{0,rdn_sinead}},
    OM_NULL_DESCRIPTOR
};

```

Step 4: Create Descriptor Lists for Attributes

The following code fragments show how the attribute lists are created for the attributes to be added to the directory.

First, initialize the public object *object_class* to contain the representation of the classes in the DIT that are common to both organizational person entries, top, person, and organizational person:

```

/* Build up an array of object identifiers for the          */
/* attributes to be added to the directory.                */
*/
*/
static OM_descriptor object_class[] = {
    OM_OID_DESC(OM_CLASS, DS_C_ATTRIBUTE),
    OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_OBJECT_CLASS),
    OM_OID_DESC(DS_ATTRIBUTE_VALUES, DS_O_TOP),
    OM_OID_DESC(DS_ATTRIBUTE_VALUES, DS_O_PERSON),
    OM_OID_DESC(DS_ATTRIBUTE_VALUES, DS_O_ORG_PERSON),
    OM_NULL_DESCRIPTOR
};

```

Next, initialize the public objects that represent the attributes to be added; *surname* and *telephone* for the distinguished name of Brendan, *surname2* and *password* for the distinguished name of Sinead:

```

static OM_descriptor telephone[] = {
    OM_OID_DESC(OM_CLASS, DS_C_ATTRIBUTE),
    OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_PHONE_NBR),
    {DS_ATTRIBUTE_VALUES, OM_S_PRINTABLE_STRING, OM_STRING("+49 89 636 0")},
    OM_NULL_DESCRIPTOR
};

static OM_descriptor surname[] = {
    OM_OID_DESC(OM_CLASS, DS_C_AVA),
    OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_SURNAME),
    {DS_ATTRIBUTE_VALUES, OM_S_PRINTABLE_STRING, OM_STRING("Moloney")},
    OM_NULL_DESCRIPTOR
};

static OM_descriptor surname2[] = {
    OM_OID_DESC(OM_CLASS, DS_C_AVA),

```

```

OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_SURNAME),
{DS_ATTRIBUTE_VALUES, OM_S_PRINTABLE_STRING, OM_STRING("Murphy")},
OM_NULL_DESCRIPTOR
};

```

```

static OM_descriptor password[] = {
OM_OID_DESC(OM_CLASS, DS_C_AVA),
OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_USER_PASSWORD),
{DS_ATTRIBUTE_VALUES, OM_S_OCTET_STRING, OM_STRING("secret")},
OM_NULL_DESCRIPTOR
};

```

Finally, initialize the public objects that represent the list of attributes to be added to the directory, *attr_list1* for the distinguished name Brendan, and *attr_list2* for the distinguished name Sinead:

```

static OM_descriptor attr_list1[] = {
OM_OID_DESC(OM_CLASS, DS_C_ATTRIBUTE_LIST),
{DS_ATTRIBUTES, OM_S_OBJECT, {0, object_class} },
{DS_ATTRIBUTES, OM_S_OBJECT, {0, surname} },
{DS_ATTRIBUTES, OM_S_OBJECT, {0, telephone} },
OM_NULL_DESCRIPTOR
};

```

```

static OM_descriptor attr_list2[] = {
OM_OID_DESC(OM_CLASS, DS_C_ATTRIBUTE_LIST),
{DS_ATTRIBUTES, OM_S_OBJECT, {0, object_class} },
{DS_ATTRIBUTES, OM_S_OBJECT, {0, surname2} },
{DS_ATTRIBUTES, OM_S_OBJECT, {0, password} },
OM_NULL_DESCRIPTOR
};

```

The *attr_list1* variable contains the public objects *surname* and *telephone*, the C representations of the attributes of the distinguished name */C=ie/O=sni/CN=Brendan* that are added to the Directory. The *attr_list2* variable contains the public objects *surname2* and *password*, the C representations of the attributes of the distinguished name */C=ie/O=sni/CN=sinead*.

Step 5: Perform the Operations

The following code fragments show the *ds_add_entry()*, *ds_list()*, and the *ds_remove_entry()* calls:

First, the two *ds_add_entry()* function calls add the attribute lists contained in *attr_list1* and *attr_list2* to the distinguished names represented by *dn_brendan* and *dn_sinead*, respectively:

```

/* Add two entries to the GDS server. */

if (ds_add_entry(bound_session, DS_DEFAULT_CONTEXT, dn_brendan, attr_list1, &invoke_id) != DS_SUCCESS)
    printf("ds_add_entry() error\n");

if (ds_add_entry(bound_session, DS_DEFAULT_CONTEXT, dn_sinead, attr_list2, &invoke_id) != DS_SUCCESS)
    printf("ds_add_entry() error\n");

```

Next, list all the subordinates of the object referenced by the distinguished name */C=ie/O=sni:*

```

if (ds_list(bound_session, DS_DEFAULT_CONTEXT, dn_sni, &result, &invoke_id) != DS_SUCCESS)
    printf("ds_list() error\n");

```

The `ds_list()` call returns the result in the private object `result` to the workspace. The components of `result` are represented by OM attributes in the OM class **DS_C_LIST_RESULT** (as shown in Figure 46 on page 156) and can only be read by a series of `om_get()` calls.

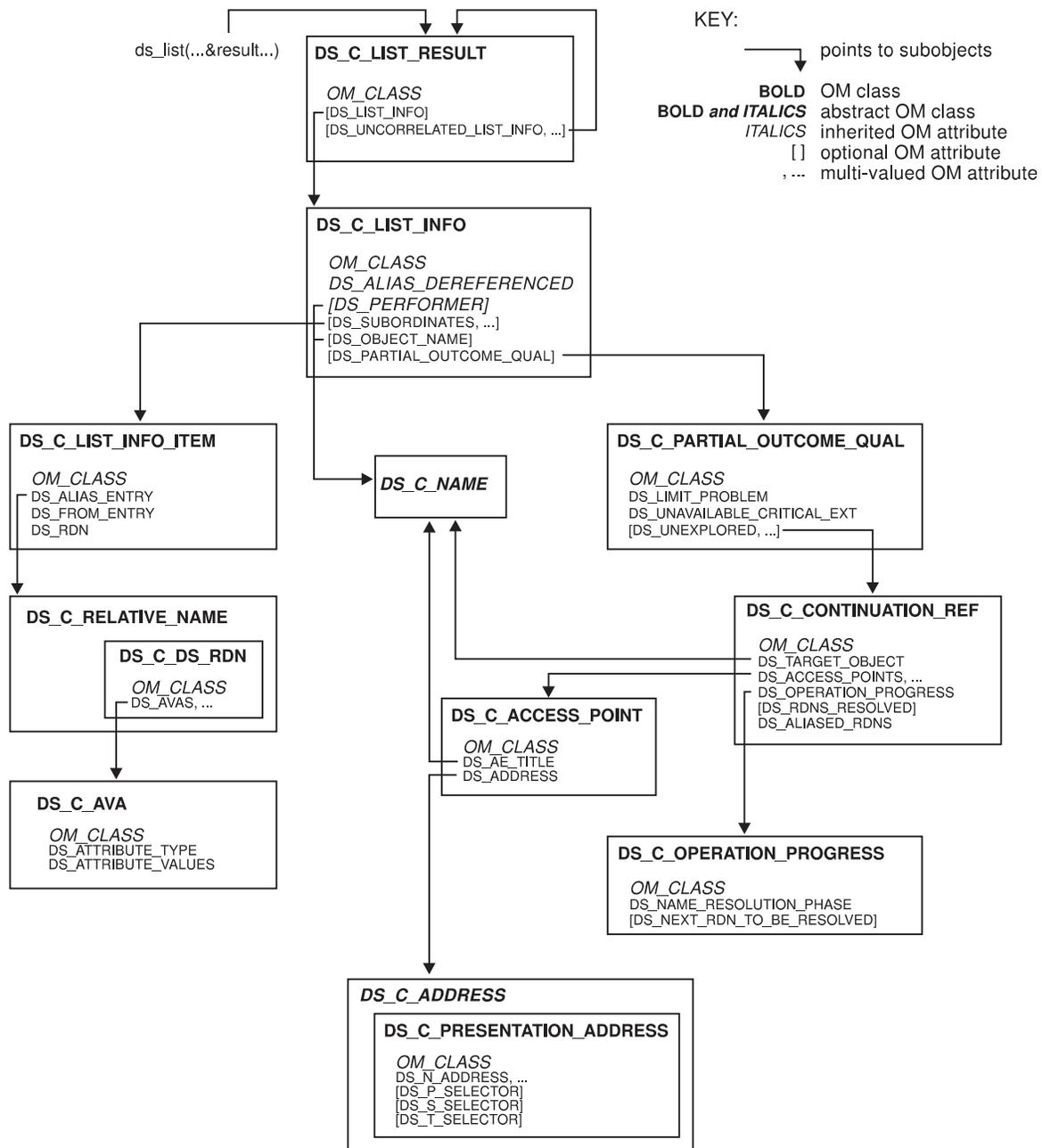


Figure 46. OM Class `DS_C_LIST_RESULT`

Finally, remove the two entries from the directory:

```

if (ds_remove_entry(bound_session, DS_DEFAULT_CONTEXT, dn_brendan, &invoke_id) != DS_SUCCESS)
    printf("ds_remove_entry() error\n");

if (ds_remove_entry(bound_session, DS_DEFAULT_CONTEXT, dn_sinead, &invoke_id) != DS_SUCCESS)
    printf("ds_remove_entry() error\n");
  
```

Return Codes

XDS API function calls return a value of type **DS_status**, the exception being **ds_initialize()** which returns a value of type **OM_workspace**. If the function is successful, then **DS_status** returns with a value of **DS_SUCCESS**. If the function does not complete successfully, then **DS_status** is set to either the error constant **DS_NO_WORKSPACE** or a private object which is a subclass of **DS_C_ERROR** (described in Chapter 11, “XDS Class Definitions” on page 241).

Chapter 7. Example Application Programs

This chapter contains two sample programs and the header file that is included (in parentheses):

- **example.c** (**example.h**)
- **teldir.c**

Most of the concepts that you will need to know to understand and use these programs have been discussed in previous chapters in Part 1, “Using the DCE Directory APIs.” The programs are arranged so that the simplest program, **example.c**, is presented first, and the most complex program, **teldir.c**, is presented last. The two programs demonstrate basic XDS and XOM API principles and concepts in operation. The **teldir.c** program is considerably more complex and uses a more sophisticated approach. It allows the user to enter values dynamically, for example, a surname and phone number.

The source code for these example programs is stored in the following directories:

- **example.c:** /usr/lpp/dce/examples/xdsxom/example
- **teldir.c:** /usr/lpp/dce/examples/xdsxom/teldir

General Programming Guidelines

Writing an application program using XDS and XOM APIs involves the following general steps before you begin coding:

1. Select the interface functions that you will need for your application and determine the parameters for the function calls.
2. Check for abstract OM classes and superclasses of objects you will manipulate for inherited OM attributes in the X/Open Directory Service API (XDS) material found in Part 1, “Using the DCE Directory APIs.”
3. Find the correct symbolic constants of appropriate packages; these can be found in the header files included with the GDS API, such as **xdsbdcp.h**.
4. Determine the error handling required.

The example.c Program

The **example.c** program uses XDS API in synchronous mode to read a telephone number or numbers from a directory entry with a given distinguished name. The program consists of the following general steps:

1. Define the required object identifier constants.
2. Declare the variables involved with Directory Service operations.
3. Build the distinguished name of **Peter Piper** as a public object for the input parameter to **ds_read()**.
4. Build a public object for the *selection* parameter to **ds_read()**.
5. Declare the variables to extract the telephone numbers using **om_get()**.
6. Initialize the directory service and get an OM workspace.
7. Pull in the required packages.
8. Bind to a default directory session.

9. Perform the read operation to extract the telephone number of a distinguished name from the directory.
10. Terminate the Directory Service session.
11. Extract the telephone numbers using a series of **om_get()** calls.
12. Release the storage occupied by private and public objects that are no longer needed.
13. Print the telephone number string.
14. Release the storage occupied by the public objects containing telephone numbers.
15. Do application-specific processing.
16. Shut down the directory service OM workspace.
17. Exit the program.

Note: The steps you follow are highlighted in bold so that you can follow the sequence as you examine the **example.c** program.

Step 1 uses the **OM_EXPORT** macro to allocate memory for the object identifier constants that represent an OM class or OM attribute. These constants are the OM attribute values that are used to build the public objects that are required as input to **ds_read()**.

Step 2 declares the variables for Directory Service operations and error handling. The **session** and **workspace** variables are required for binding a session to a server and creating a workspace into which **ds_read()** can deposit the results of the read operation on the directory.

The **result** variable is a pointer that is returned by **ds_read()** to the workspace. The information stored in **result** is in implementation-specific private format not accessible directly by the application program. Subsequent **om_get()** calls extract the telephone numbers requested by the program from **result** and store the information in the variable **telephones** (declared in **Step 5**).

The **error** and **return_code** variables are used by the program for error handling. The **error** variable is used for processing the return code from XDS API function calls. The **return_code** is used by the error handling in header file **example.h** for processing return codes from **om_get()** function calls.

Step 3 builds the public object representing the distinguished name of **Peter Piper**. The program uses statically defined public objects to demonstrate the basic principles of building public objects. The name built differs depending on whether a DNS or an X.500-style cell name is in effect. (This is controlled by the **CELLNAME_TYPE** pre-processor directive.) A more sophisticated approach is presented in another example program, **teldir.c**. The **teldir.c** program dynamically defines a public object from a user-supplied name in DCE string format.

In this program (**example.c**), the process starts with the definition of an array of descriptor lists as AVAs. These AVAs are public objects that are included in the definition of RDNs. The RDNs, in turn, are included in the distinguished name of **Peter Piper** stored in **name**.

Using the same method of static definition, **Step 4** defines the **DS_C_ENTRY_INFO_SELECTION** public object and stores it in the variable, **selection**. The **name** and **selection** variables are required as input parameters to **ds_read()**. This process is described in detail in Chapter 6, "XDS Programming" on page 137.

Step 5 declares the variables required by **om_get()** to extract the telephone numbers from **result**. The **entry_list**, **attributes_list**, and **telephone_list** variables are of type **OM_type** and are initialized to the values of OM attribute types: **DS_ENTRY**, **DS_ATTRIBUTES**, and **DS_ATTRIBUTE_VALUES**, respectively. **DS_ENTRY** contains the selected list of entries; **DS_ATTRIBUTES** contains the selected list of attribute types; and **DS_ATTRIBUTE_VALUES** contains the actual values of the telephone numbers.

The **entry**, **attributes**, and **telephones** variables are of type **OM_public_object** because they store the output parameters of **om_get()**. The **om_get()** call makes these objects available to the application program as public object data types. The program must remove layers of objects and subobjects to get at the actual string data values of the telephone numbers. Another example program, **teldir2.c**, provides an alternative to using **om_get** to remove layers of objects: it uses the XOM Convenience routines.

The **telephones** variable contains the actual string values of the telephone numbers. It is a descriptor in the array of descriptors that make up the public object that contains the actual string data that the program wants to extract from the directory.

Step 6 initializes the Directory Service and gets an OM workspace in which **ds_read()** deposits the result of the read operation.

Step 7 pulls in the Basic Directory Contents Package into the program because it contains features that are required by the program and not included in the default package (the Directory Service Package).

Step 8 binds the session to the default session. An application program can bind with a specifically tailored session object using an instance of OM class **DS_C_SESSION**. In most cases, however, it is sufficient to use the constant **DS_DEFAULT_SESSION**. **DS_DEFAULT_SESSION** uses the default values of the OM class **DS_C_SESSION**.

Step 9 performs the read operation and deposits the result in the workspace in **result**. The **result** variable is one of the input parameters for the **om_get()** function call. The **session** and **DS_DEFAULT_CONTEXT** constant are the *session* and *context* parameters required to be present in the **ds_read()** function call.

The **name** holds the public object representing the distinguished name of **Peter Piper**, and the **selection** variable contains the public object indicating which attributes and values are to be selected by the read operation from the entry. The *invoke_id* parameter is not used by the DCE-implementation of XDS API and is ignored.

Step 10 terminates the directory session.

Step 11 uses a series of **om_get()** calls to extract the telephone numbers. The first **om_get()** extracts the information about the entry from **result** and puts it in **entry**. The second **om_get()** extracts the attribute types from **entry** and puts them in **attributes**. The third **om_get()** extracts the actual values of the telephone numbers from **attributes** and puts them in **telephones**. The **telephones** variable contains the string data values of the telephone numbers.

Step 12 releases the storage occupied by the private and public objects that are no longer needed. The program has the data values in **telephones** that it needs to continue processing.

Step 13 prints out each telephone number associated with the distinguished name **Peter Piper** in the directory, or returns an error message if the number is not in the correct format. It checks for an attribute with type **DS_ATTRIBUTE_VALUES**.

Step 14 releases the storage occupied by **telephones** because it is no longer needed.

In **Step 15** shows where application-specific processing can occur.

In **Step 16** a **ds_shutdown()** is issued that shuts down the interface established by **ds_initialize()**.

Step 17 continues processing and exits.

The example.c Code

The following code is a listing of the **example.c** program:

```
/*
 *      Default this example to use a DNS style cell name.
 */

#define DNS_TYPE 1
#define GDS_TYPE 2
#ifndef CELLNAME_TYPE
    #define CELLNAME_TYPE    DNS_TYPE
#endif

#if CELLNAME_TYPE == DNS_TYPE
    #ifndef DNS_CELLNAME
        #define DNS_CELLNAME    "cellname"
    #endif
#else
    #ifndef GDS_CELLNAME_C
        #define GDS_CELLNAME_C    "US"
    #endif
    #ifndef GDS_CELLNAME_O
        #define GDS_CELLNAME_O    "Acme Pepper Co"
    #endif
    #ifndef GDS_CELLNAME_OU
        #define GDS_CELLNAME_OU    "Research"
    #endif
#endif

#ifndef CDS_DIR_NAME
    #define CDS_DIR_NAME        "PhoneBook"
#endif
#ifndef CDS_OBJ_NAME
    #define CDS_OBJ_NAME        "Peter Piper"
#endif

/*
 * sample application that uses XDS in synchronous mode
 *
 * This program reads the telephone number(s) of a given target name.
 */

#ifdef THREADSAFE
    #include <pthread.h>
#endif
#include <stdio.h>

#include <xom.h>
#include <xds.h>
#include <xdsbdcp.h>
#include <xds cds.h>

#include "example.h" /* possible Error Handling header */

/* Step 1
 *
```

```

* Define necessary Object Identifier constants
*/
OM_EXPORT(DSX_TYPELESS_RDN)
#if CELLNAME_TYPE == GDS_TYPE
    OM_EXPORT(DS_A_COUNTRY_NAME)
    OM_EXPORT(DS_A_ORG_NAME)
    OM_EXPORT(DS_A_ORG_UNIT_NAME)
#endif
OM_EXPORT(DS_A_PHONE_NBR)
OM_EXPORT(DS_C_AVA)
OM_EXPORT(DS_C_DS_DN)
OM_EXPORT(DS_C_DS_RDN)
OM_EXPORT(DS_C_ENTRY_INFO_SELECTION)

/* Step 2 */

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

    static DS_feature bdcg_package[] = {
    { OM_STRING(OMP_O_DS_BASIC_DIR_CONTENTS_PKG), OM_TRUE },
    { { (OM_uint32)0, (void *)0 }, OM_FALSE },
    };

/* Step 3
*
* Public Object ("Descriptor List") for Name argument to ds_read().
* Build the Distinguished-Name of the form
*      ../../<cellname>/<cds_dir_name>/<cds_obj_name>
*/

#if CELLNAME_TYPE == DNS_TYPE

    static OM_descriptor  cellname[] = {
    OM_OID_DESC(OM_CLASS, DS_C_AVA),
    OM_OID_DESC(DS_ATTRIBUTE_TYPE, DSX_TYPELESS_RDN),
    { DS_ATTRIBUTE_VALUES, OM_S_PRINTABLE_STRING | OM_S_LOCAL_STRING,
      OM_STRING(DNS_CELLNAME) },
    OM_NULL_DESCRIPTOR
    };

    static OM_descriptor  directory_name[] = {
    OM_OID_DESC(OM_CLASS, DS_C_AVA),
    OM_OID_DESC(DS_ATTRIBUTE_TYPE, DSX_TYPELESS_RDN),
    { DS_ATTRIBUTE_VALUES, OM_S_TELETEX_STRING | OM_S_LOCAL_STRING,
      OM_STRING(CDS_DIR_NAME) },
    OM_NULL_DESCRIPTOR
    };

    static OM_descriptor  object_name[] = {
    OM_OID_DESC(OM_CLASS, DS_C_AVA),

```

```

OM_OID_DESC(DS_ATTRIBUTE_TYPE, DSX_TYPELESS_RDN),
{ DS_ATTRIBUTE_VALUES, OM_S_TELETEX_STRING | OM_S_LOCAL_STRING,
  OM_STRING(CDS_OBJ_NAME) },
OM_NULL_DESCRIPTOR
};

```

```

    static OM_descriptor    rdn1[] = {
OM_OID_DESC(OM_CLASS, DS_C_DS_RDN),
{ DS_AVAS, OM_S_OBJECT, { 0, cellname } },
OM_NULL_DESCRIPTOR
};

```

```

    static OM_descriptor    rdn2[] = {
OM_OID_DESC(OM_CLASS, DS_C_DS_RDN),
{ DS_AVAS, OM_S_OBJECT, { 0, directory_name } },
OM_NULL_DESCRIPTOR
};

```

```

    static OM_descriptor    rdn3[] = {
OM_OID_DESC(OM_CLASS, DS_C_DS_RDN),
{ DS_AVAS, OM_S_OBJECT, { 0, object_name } },
OM_NULL_DESCRIPTOR
};

```

```

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

```

#else

```

    static OM_descriptor    country[] = {
OM_OID_DESC(OM_CLASS, DS_C_AVA),
OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_COUNTRY_NAME),
{ DS_ATTRIBUTE_VALUES, OM_S_PRINTABLE_STRING | OM_S_LOCAL_STRING,
  OM_STRING(GDS_CELLNAME_C) },
OM_NULL_DESCRIPTOR
};

```

```

    static OM_descriptor    organization[] = {
OM_OID_DESC(OM_CLASS, DS_C_AVA),
OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_ORG_NAME),
{ DS_ATTRIBUTE_VALUES, OM_S_TELETEX_STRING | OM_S_LOCAL_STRING,
  OM_STRING(GDS_CELLNAME_O) },
OM_NULL_DESCRIPTOR
};

```

```

    static OM_descriptor    organizational_unit[] = {
OM_OID_DESC(OM_CLASS, DS_C_AVA),
OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_ORG_UNIT_NAME),
{ DS_ATTRIBUTE_VALUES, OM_S_TELETEX_STRING | OM_S_LOCAL_STRING,
  OM_STRING(GDS_CELLNAME_OU) },
OM_NULL_DESCRIPTOR
};

```

```

    static OM_descriptor    directory_name[] = {
OM_OID_DESC(OM_CLASS, DS_C_AVA),
OM_OID_DESC(DS_ATTRIBUTE_TYPE, DSX_TYPELESS_RDN),
{ DS_ATTRIBUTE_VALUES, OM_S_TELETEX_STRING | OM_S_LOCAL_STRING,
  OM_STRING(CDS_DIR_NAME) },

```

```

OM_NULL_DESCRIPTOR
};
    static OM_descriptor    object_name[] = {
OM_OID_DESC(OM_CLASS, DS_C_AVA),
OM_OID_DESC(DS_ATTRIBUTE_TYPE, DSX_TYPELESS_RDN),
{ DS_ATTRIBUTE_VALUES, OM_S_TELETEX_STRING | OM_S_LOCAL_STRING,
    OM_STRING(CDS_OBJ_NAME) },
OM_NULL_DESCRIPTOR
};

    static OM_descriptor    rdn1[] = {
OM_OID_DESC(OM_CLASS, DS_C_DS_RDN),
{ DS_AVAS, OM_S_OBJECT, { 0, country } },
OM_NULL_DESCRIPTOR
};
    static OM_descriptor    rdn2[] = {
OM_OID_DESC(OM_CLASS, DS_C_DS_RDN),
{ DS_AVAS, OM_S_OBJECT, { 0, organization } },
OM_NULL_DESCRIPTOR
};
    static OM_descriptor    rdn3[] = {
OM_OID_DESC(OM_CLASS, DS_C_DS_RDN),
{ DS_AVAS, OM_S_OBJECT, { 0, organizational_unit } },
OM_NULL_DESCRIPTOR
};
    static OM_descriptor    rdn4[] = {
OM_OID_DESC(OM_CLASS, DS_C_DS_RDN),
{ DS_AVAS, OM_S_OBJECT, { 0, directory_name } },
OM_NULL_DESCRIPTOR
};
    static OM_descriptor    rdn5[] = {
OM_OID_DESC(OM_CLASS, DS_C_DS_RDN),
{ DS_AVAS, OM_S_OBJECT, { 0, object_name } },
OM_NULL_DESCRIPTOR
};

    OM_descriptor    name[] = {
OM_OID_DESC(OM_CLASS, DS_C_DS_DN),
{ DS_RDNS, OM_S_OBJECT, { 0, rdn1 } },
{ DS_RDNS, OM_S_OBJECT, { 0, rdn2 } },
{ DS_RDNS, OM_S_OBJECT, { 0, rdn3 } },
{ DS_RDNS, OM_S_OBJECT, { 0, rdn4 } },
{ DS_RDNS, OM_S_OBJECT, { 0, rdn5 } },
OM_NULL_DESCRIPTOR
};

#endif

/* Step 4 */

/*
 *
 * Public Object ("Descriptor List") for
 * for Entry-Information-Selection argument to ds_read().
 */
OM_descriptor selection[] = {
    OM_OID_DESC(OM_CLASS, DS_C_ENTRY_INFO_SELECTION),

```

```

    { DS_ALL_ATTRIBUTES, OM_S_BOOLEAN, { OM_FALSE, NULL } },
    OM_OID_DESC(DS_ATTRIBUTES_SELECTED, DS_A_PHONE_NBR),
    { DS_INFO_TYPE,OM_S_ENUMERATION, { DS_TYPES_AND_VALUES,NULL } },
    OM_NULL_DESCRIPTOR
};

/* Step 5 */

/*
 * variables to extract the telephone number(s)
 */
OM_type      entry_list[]      = { DS_ENTRY, 0 };
OM_type      attributes_list[] = { DS_ATTRIBUTES, 0 };
OM_type      telephone_list[] = { DS_ATTRIBUTE_VALUES, 0 };
OM_public_object entry;
OM_public_object attributes;
OM_public_object telephones;
OM_descriptor *telephone;      /* current phone number */

/*
 * Perform the Directory operations:
 * (1) Initialize the Directory Service and get an OM workspace
 * (2) bind a default directory session.
 * (3) read the telephone number of "name".
 * (4) terminate the directory session.
 */

/* Step 6 */
CHECK_DS_CALL((OM_object) !(workspace=ds_initialize()));

/* Step 7 */
CHECK_DS_CALL(ds_version(bdcp_package, workspace));

/* Step 8 */
CHECK_DS_CALL(ds_bind(DS_DEFAULT_SESSION, workspace, &session));

/* Step 9 */
CHECK_DS_CALL(ds_read(session, DS_DEFAULT_CONTEXT, name, selection,
                      &result, &invoke_id));

/*
 * NOTE: should check here for Attribute-Error (no-such-attribute)
 * in case the "name" doesn't have a telephone.
 * Then for all other cases call error_handler
 */

/* Step 10 */
CHECK_DS_CALL(ds_unbind(session));

/* Step 11 */

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

```

```

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

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

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

```

/* Step 12 */

```

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

```

/* Step 13 */

```

for (telephone = telephones;
     telephone->type == DS_ATTRIBUTE_VALUES;
     telephone++)
{
    if ( (telephone->type != DS_ATTRIBUTE_VALUES)
        || ( (telephone->syntax & OM_S_SYNTAX) != OM_S_PRINTABLE_STRING
            && (telephone->syntax & OM_S_SYNTAX) != OM_S_TELETEX_STRING
        )
    )
    {
        (void) fprintf(stderr, "malformed telephone number\n");
        exit(EXIT_FAILURE);
    }

    telephone->value.string.length,
    telephone->value.string.elements);
}

```

/* Step 14 */

```

CHECK_OM_CALL(om_delete(telephones));

```

/* Step 15 */

```

/* more application-specific processing can occur here...
 */

```

```

/* Step 16 */

/* We're done with all service generated public objects
 * and done with the workspace
 */
CHECK_DS_CALL(ds_shutdown(workspace));

/* Step 17 */

/* ... and finally exit. */
exit(EXIT_SUCCESS);
}

```

Error Handling

The **example.c** program includes the header file **example.h** for error handling of XDS and XOM API function calls. The **example.h** program contains two error handling functions: **CHECK_DS_CALL** for handling XDS API errors and **CHECK_OM_CALL** for handling XOM API errors. Note that **CHECK_DS_CALL** and **CHECK_OM_CALL** are created specifically for **example.c** and are not part of the XDS or XOM APIs. They are included to demonstrate a possible method for error handling.

XDS and XOM API functions return a status code. In **example.c**, **error** contains the status code for XDS API functions. If the call is successful, the function returns **DS_SUCCESS**. Otherwise, one of the error codes described in “XDS Errors” on page 241 is returned.

The **return_code** variable contains the status code for XOM API functions. If the call is successful, the function returns **OM_SUCCESS**. Otherwise, one of the error codes described in Chapter 18, “XOM Service Interface” on page 331 is returned.

The contents of **example.h** are as follows:

```

#ifndef _EXAMPLE_H
#define _EXAMPLE_H

#ifdef __cplusplus
extern "C" {
#endif

/*
 * define some convenient exit codes
 */

#define EXIT_FAILURE 1
#define EXIT_SUCCESS 0

/*
 * declare an error handling function and
 * an error checking macro for DS
 */

void handle_ds_error(DS_status error);

#define CHECK_DS_CALL(function_call) \
    error = (function_call) \
    if (error != DS_SUCCESS) \

```

```

        handle_ds_error(error);

/*
 * declare an error handling function and
 * an error checking macro for OM
 */

void handle_om_error(OM_return_code return_code);

#define CHECK_OM_CALL(function_call) \
        return_code = (function_call) \
        if (return_code != OM_SUCCESS) \
            handle_om_error(return_code);

/*
 * the error handling code
 *
 * NOTE: any errors arising in these functions are ignored.
 */

void handle_ds_error(DS_status error)
{
    (void) fprintf(stderr, "DS error has occurred\n");

    (void) om_delete((OM_object) error);

    /* At this point, the error has been reported and storage cleaned
     * up, so the handler could return to the main program now for it
     * to take recovery action. But we choose the simple option ...
     */

    exit(EXIT_FAILURE);
}

void handle_om_error(OM_return_code return_code)
{
    (void) fprintf(stderr, "OM error %d has occurred\n", return_code);

    /* At this point, the error has been reported and storage cleaned up,
     * so the handler could return to the main program now for it to take
     * recovery action. But we choose the simple option ...
     */

    exit(EXIT_FAILURE);
}

#ifdef __cplusplus
}
#endif

#endif /* _EXAMPLE_H */

```

The teldir.c Program

The sample program **teldir.c** permits a user to add, read, or delete entries in a CDS or GDS namespace in any local or remote DCE cell (assuming that proper permissions have been granted to the user of the program). The entry consists of a person's surname and phone number. Each entry is of class Organizational Person.

The program uses predefined static XDS public objects that are never altered, and partially defined static XDS public objects so the user can dynamically enter values for the surname and phone number. It also uses dynamic XDS public objects that are created and filled only as needed using the **stringToXdsName** function. These techniques are a departure from the ones used in the first sample program where all objects are predefined.

Predefined Static Public Objects

The predefined static object classes and attributes are shown in the following code fragment:

```
/*
 * To hold the attributes we want to attach to the name being added.
 * One attribute is the class of the object (DS_O_ORG_PERSON), the
 * rest of the attributes are the surname (required for all objects
 * of class DS_O_ORG_PERSON) and phone number. In addition, we need
 * an object to hold all this information to pass it into ds_add_entry().
 */
static OM_descriptor xdsObjectClass[] = {

    /* This object is an attribute--an object class. */
    OM_OID_DESC( OM_CLASS,          DS_C_ATTRIBUTE ),
    OM_OID_DESC( DS_ATTRIBUTE_TYPE,  DS_A_OBJECT_CLASS ),

    /* List the class. It will inherit its superclasses */
    OM_OID_DESC( DS_ATTRIBUTE_VALUES, DS_O_ORG_PERSON ),

    /* Null terminator */
    OM_NULL_DESCRIPTOR
};
static OM_descriptor xdsAttributesToAdd[] = {

    /* This object is an attribute list. */
    OM_OID_DESC( OM_CLASS, DS_C_ATTRIBUTE_LIST ),

    /* These are "pointers" to the attributes in the list. */
    { DS_ATTRIBUTES, OM_S_OBJECT, { 0, xdsObjectClass } },
    { DS_ATTRIBUTES, OM_S_OBJECT, { 0, xdsSurname } },
    { DS_ATTRIBUTES, OM_S_OBJECT, { 0, xdsPhoneNum } },

    /* Null terminator */
    OM_NULL_DESCRIPTOR
};

/*
 * To hold the list of attributes we want to read.
 */
static OM_descriptor xdsAttributeSelection[] = {

    /* This is an entry information selection. */
```

```

OM_OID_DESC( OM_CLASS, DS_C_ENTRY_INFO_SELECTION ),

/* Get all attributes. */
{ DS_ALL_ATTRIBUTES, OM_S_BOOLEAN, OM_TRUE },

/* These are the ones we want to read. */
OM_OID_DESC( DS_ATTRIBUTES_SELECTED, DS_A_SURNAME ),
OM_OID_DESC( DS_ATTRIBUTES_SELECTED, DS_A_PHONE_NBR ),

/* Give us both the types and their values. */
{ DS_INFO_TYPE, OM_S_ENUMERATION, { DS_TYPES_AND_VALUES, NULL } },

/* Null terminator */
OM_NULL_DESCRIPTOR
};

```

Partially Defined Static Public Objects

The program partially defines static XDS objects with placeholders so that values for the surname and telephone number entered by the user can be added later, as shown in the following code fragment:

```

static OM_descriptor xdsSurname[] = {

    /* This object is an attribute--a surname. */
    OM_OID_DESC( OM_CLASS,          DS_C_ATTRIBUTE ),
    OM_OID_DESC( DS_ATTRIBUTE_TYPE, DS_A_SURNAME ),

    /* No default--so we need a place holder for the actual surname. */
    OM_NULL_DESCRIPTOR,

    /* Null terminator */
    OM_NULL_DESCRIPTOR
};

static OM_descriptor xdsPhoneNum[] = {

    /* This object is an attribute--a telephone number. */
    OM_OID_DESC( OM_CLASS,          DS_C_ATTRIBUTE ),
    OM_OID_DESC( DS_ATTRIBUTE_TYPE, DS_A_PHONE_NBR ),

    /* By default, phone numbers are unlisted. If the user specifies */
    /* an actual phone number, it will go into this position.          */
    { DS_ATTRIBUTE_VALUES, OM_S_PRINTABLE_STRING|OM_S_LOCAL_STRING, OM_STRING( "unlisted" ) },

    /* Null terminator */
    OM_NULL_DESCRIPTOR
};

```

The program prompts the user for the surname of the person whose number will be changed and uses the **FILL_OMD_STRING** macro to fill in values, as shown in the following code fragment:

```

if ( operation == 'a' ) { /* add operation requires additional input */
    /*
     * Get the person's real name from the user and place it in the
     * XDS object already defined at the top of the program (xdsSurname).
     * We are requiring a name, so we will loop until we get one.

```

```

    */
do {
    printf( "What is this person's surname? " );
    gets( newSurname );
} while ( *newSurname == '\0' );
FILL_OMD_STRING( xdsSurname, 2, DS_ATTRIBUTE_VALUES,
                OM_S_TELETEX_STRING|OM_S_LOCAL_STRING, newSurname )

```

Dynamically Defined Public Objects

The program uses the function **stringToXdsName** to convert the DCE name entered by a user into an XDS name object of OM class **DS_C_DS_DN**, which is the representation of a distinguished name. In the other example program, **example.c**, arrays of descriptor lists are statically declared to represent the AVAs and RDNs that make up the public object that represents a distinguished name. The function **stringToXdsName** parses the DCE name and dynamically converts it to a public object.

For example, the following code fragment shows how space for a **DS_C_AVA** object is allocated and its entries are filled using the **FILL_OMD_XOM_STRING** and **FILL_OMD_NULL** macros:

```

/*
 * Allocate space for a DS_C_AVA object and fill in its entries:
 *   DS_C_AVA class identifier
 *   AVA's type
 *   AVA's value
 *   null terminator
 */
ava = (OM_descriptor *)malloc( sizeof(OM_descriptor) * 4 );
if( ava == NULL ) /* malloc() failed */
    return OM_MEMORY_INSUFFICIENT;
FILL_OMD_XOM_STRING( ava, 0, OM_CLASS, OM_S_OBJECT_IDENTIFIER_STRING,
                    DS_C_AVA )
splitNamePiece( start, &type, &value );
FILL_OMD_XOM_STRING( ava, 1, DS_ATTRIBUTE_TYPE, OM_S_OBJECT_IDENTIFIER_STRING,
                    type )
FILL_OMD_STRING( ava, 2, DS_ATTRIBUTE_VALUES, OM_S_PRINTABLE_STRING|OM_S_LOCAL_STRING,
                value )
FILL_OMD_NULL( ava, 3 )

```

The program uses the same method to build the RDNs that make up the distinguished name. The distinguished name is **NULL** terminated using the **FILL_OMD_NULL** macro and the location of the new public object is provided for the calling routine (main) in the pointer *xdsNameObj*, as shown in the following code fragment:

```

/* Add the DS_C_RDN object to the DS_C_DS_DN object. */
FILL_OMD_STRUCT( dsdn, index, DS_RDNS, OM_S_OBJECT, rdn )
}

/*
 * Null terminate the DS_C_DS_DN, tell the calling routine
 * where to find it, and return.
 */
FILL_OMD_NULL( dsdn, index )
*xdsNameObj = dsdn;
return( OM_SUCCESS );

} /* end stringToXdsName() */

```

Main Program Procedural Steps

The program consists of the following general steps:

1. Examine the command-line argument to determine the type of operation (read, add, or delete entry) that the user wants to perform.
2. Initialize a workspace.
3. Pull in the packages with the required XDS features.
4. Prompt the user for the name entry on which the operation will be performed.
5. Convert the DCE-formatted user input string to an XDS object name.
6. Bind (without credentials) to the default server.
7. Perform the requested operation (read, add, or delete entry).
8. Unbind from the server.
9. Shutdown the workspace, releasing resources back to the system.

Note: The steps that follow are highlighted in bold so that you can follow the sequence as you examine the `teldir.c` program.

Step 1 simply involves determining which of the three options: **r** (read), **a** (add), or **d** (delete) the user has entered. **Step 2** initializes a workspace, an operation required by XDS API for every application program. **Step 3** is required because additional features not present in the Directory Service Package need to be used by the application program. An additional package, the Basic Directory Contents Package, is defined in `featureList`, as a static XDS object earlier in the program.

In **Step 4**, the user is prompted for the DCE-formatted name, which is the distinguished name of the person on whose telephone number the operation is to be performed. The name must be a fully or partially qualified name that begins with either the `/...` or `/.` prefix. An example of a fully qualified, or global, name is `/.../C=de/O=sni/OU=ap/CN=klaus`. An example of a partially qualified, or cell, name is `/./brad/sni/com`. Additional information is requested in **Step 5** if the user requests an add operation.

Step 5 converts the DCE-formatted name to an XDS object name (public object) using the `stringToXdsName` function call. This function builds an XDS public object that represents the distinguished name entered by the user.

Step 6 binds the session to the default server without credentials; user name and password are not required.

In **Step 7**, the requested operation is performed using XDS API function calls. For an add operation, `ds_add_entry()` is performed; for a read operation, `ds_read()` is performed; and for a delete operation, `ds_remove_entry()` is performed. The read operation requires a series of XOM API `om_get()` function calls to extract the surname and phone number from the workspace. This is performed inside the `extractValue()` function. (For a detailed description of the XDS and XOM API function calls, refer to *z/OS DCE Application Development Reference*.)

Step 8 and **Step 9** are required for every XDS API application program in order to clean up before the program exits. The session is unbound from the server, and the public and private objects are released to the system that provided the memory allocated for them.

The teldir.c Code

The following is a listing of the file **teldir.c**:

```
/*
 * This sample program behaves like a simple telephone directory.
 * It permits a user to add, read or delete entries in a GDS
 * namespace or to a CDS namespace in any local or remote DCE cell
 * (assuming that permissions are granted by the ACLs).
 *
 * Each entry is of class Organizational-Person and simply contains
 * a person's surname and their phone number.
 *
 * The addition of an entry is followed by a read to verify that the
 * information was entered properly.
 *
 * All valid names should begin with one of the following symbols:
 *   /...      Fully qualified name (from global root).
 *             e.g. /.../C=de/O=sni/OU=ap/CN=klaus
 *
 *   /.:      Partially qualified name (from local cell root).
 *             e.g. /.:/brad/sni/com
 *
 * This program demonstrates the following techniques:
 * - Using completely static XDS public objects (pre-defined at the top
 *   of the program and never altered). See xdsObjectClass,
 *   xdsAttributesToAdd, and xdsAttributeSelection below.
 * - Using partially static XDS public objects (pre-defined at the top
 *   of the program but altered later). See xdsSurname and xdsPhoneNum
 *   below. See also the macros whose names begin with "FILL_OMD_".
 * - Using dynamic XDS public objects (created and filled in only as
 *   needed). See the function stringToXdsName() below.
 * - Parsing DCE-style names and converting them into XDS objects. See
 *   the function stringToXdsName() below.
 * - Getting the value of an attribute from an object read from the
 *   namespace using ds_read(). See the function extractValue() below.
 * - Getting the numeric value of an error (type DS_status) returned by
 *   one of the XDS calls. See the function handleDSError() below.
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <xom.h>
#include <xds.h>
#include <xdsbdcp.h>
#include <xdsqds.h>
#include <xdseds.h>

OM_EXPORT( DS_A_COMMON_NAME )
OM_EXPORT( DS_A_COUNTRY_NAME )
OM_EXPORT( DS_A_LOCALITY_NAME )
OM_EXPORT( DS_A_OBJECT_CLASS )
OM_EXPORT( DS_A_ORG_UNIT_NAME )
OM_EXPORT( DS_A_ORG_NAME )
OM_EXPORT( DS_A_SURNAME )
OM_EXPORT( DS_A_PHONE_NBR )
```

```

OM_EXPORT( DS_A_TITLE )
OM_EXPORT( DS_C_ATTRIBUTE )
OM_EXPORT( DS_C_ATTRIBUTE_LIST )
OM_EXPORT( DS_A_STATE_OR_PROV_NAME )
OM_EXPORT( DS_C_AVA )
OM_EXPORT( DS_C_DS_DN )
OM_EXPORT( DS_C_DS_RDN )
OM_EXPORT( DS_C_ENTRY_INFO_SELECTION )
OM_EXPORT( DS_O_ORG_PERSON )
OM_EXPORT( DS_O_PERSON )
OM_EXPORT( DS_O_TOP )
OM_EXPORT( DSX_TYPELESS_RDN ) /* For "typeless" pieces of a name, as */
                               /* found in cells with bind-style names */
                               /* and in the CDS namespace. */

#define MAX_NAME_LEN 1024

/* These values can be found in the Chapter
   "Directory Class Definitions". */
/* (One byte must be added for the null terminator.) */
#define MAX_PHONE_LEN 33
#define MAX_SURNAME_LEN 66

/*****
 * Macros for help filling in static XDS objects.
 *****/
/* Put NULL value (equivalent to OM_NULL_DESCRIPTOR) in object */
#define FILL_OMD_NULL( desc, index ) \
    desc[index].type = OM_NO_MORE_TYPES; \
    desc[index].syntax = OM_S_NO_MORE_SYNTAXES; \
    desc[index].value.object.padding = 0; \
    desc[index].value.object.object = OM_ELEMENTS_UNSPECIFIED;

/* Put C-style (null-terminated) string in object */
#define FILL_OMD_STRING( desc, index, typ, syntx, val ) \
    desc[index].type = typ; \
    desc[index].syntax = syntx; \
    desc[index].value.string.length = strlen( val ); \
    desc[index].value.string.elements = val;

/* Put XOM string in object */
#define FILL_OMD_XOM_STRING( desc, index, typ, syntx, val ) \
    desc[index].type = typ; \
    desc[index].syntax = syntx; \
    desc[index].value.string.length = val.length; \
    desc[index].value.string.elements = val.elements;

/* Put other value in object */
#define FILL_OMD_STRUCT( desc, index, typ, syntx, val ) \
    desc[index].type = typ; \
    desc[index].syntax = syntx; \
    desc[index].value.object.padding = 0; \
    desc[index].value.object.object = val;

/*****
 * Static XDS objects.
 *****/
/*

```

```

* To identify which packages we need for this program. We only need
* the basic package because we are not doing anything fancy with
* session parameters, etc.
*/
DS_feature featureList[] = {
    { OM_STRING(OMP_O_DS_BASIC_DIR_CONTENTS_PKG), OM_TRUE },
    { 0 }
};

/*
* To hold the attributes we want to attach to the name being added.
* One attribute is the class of the object (DS_O_ORG_PERSON), the
* rest of the attributes are the surname (required for all objects
* of class DS_O_ORG_PERSON) and phone number. In addition, we need
* an object to hold all this information to pass it into ds_add_entry().
*/
static OM_descriptor xdsObjectClass[] = {

    /* This object is an attribute--an object class. */
    OM_OID_DESC( OM_CLASS,          DS_C_ATTRIBUTE ),
    OM_OID_DESC( DS_ATTRIBUTE_TYPE, DS_A_OBJECT_CLASS ),

    /* List the class. It will inherit its superclasses */
    OM_OID_DESC( DS_ATTRIBUTE_VALUES, DS_O_ORG_PERSON ),

    /* Null terminator */
    OM_NULL_DESCRIPTOR
};

static OM_descriptor xdsSurname[] = {

    /* This object is an attribute--a surname. */
    OM_OID_DESC( OM_CLASS,          DS_C_ATTRIBUTE ),
    OM_OID_DESC( DS_ATTRIBUTE_TYPE, DS_A_SURNAME ),

    /* No default--so we need a place holder for the actual surname. */
    OM_NULL_DESCRIPTOR,

    /* Null terminator */
    OM_NULL_DESCRIPTOR
};

static OM_descriptor xdsPhoneNum[] = {

    /* This object is an attribute--a telephone number. */
    OM_OID_DESC( OM_CLASS,          DS_C_ATTRIBUTE ),
    OM_OID_DESC( DS_ATTRIBUTE_TYPE, DS_A_PHONE_NBR ),

    /* By default, phone numbers are unlisted. If the user specifies */
    /* an actual phone number, it will go into this position. */
    { DS_ATTRIBUTE_VALUES, OM_S_PRINTABLE_STRING | OM_S_LOCAL_STRING,
      OM_STRING( "unlisted" ) },

    /* Null terminator */
    OM_NULL_DESCRIPTOR
};

static OM_descriptor xdsAttributesToAdd[] = {

```

```

/* This object is an attribute list. */
OM_OID_DESC( OM_CLASS, DS_C_ATTRIBUTE_LIST ),

/* These are "pointers" to the attributes in the list. */
{ DS_ATTRIBUTES, OM_S_OBJECT, { 0, xdsObjectClass } },
{ DS_ATTRIBUTES, OM_S_OBJECT, { 0, xdsSurname } },
{ DS_ATTRIBUTES, OM_S_OBJECT, { 0, xdsPhoneNum } },

/* Null terminator */
OM_NULL_DESCRIPTOR
};

/*
 * To hold the list of attributes we want to read.
 */
static OM_descriptor xdsAttributeSelection[] = {

/* This is an entry information selection. */
OM_OID_DESC( OM_CLASS, DS_C_ENTRY_INFO_SELECTION ),

/* Get all attributes. */
{ DS_ALL_ATTRIBUTES, OM_S_BOOLEAN, OM_TRUE },

/* These are the ones we want to read. */
OM_OID_DESC( DS_ATTRIBUTES_SELECTED, DS_A_SURNAME ),
OM_OID_DESC( DS_ATTRIBUTES_SELECTED, DS_A_PHONE_NBR ),

/* Give us both the types and their values. */
{ DS_INFO_TYPE, OM_S_ENUMERATION, DS_TYPES_AND_VALUES },

/* Null terminator */
OM_NULL_DESCRIPTOR
};

/*****
 * showUsage()
 *   Display "usage" information.
 *****/
void
showUsage(
    char *    cmd          /* In--Name of command being called */
)
{
    fprintf( stderr, "\nusage:  %s [option]\n\n", cmd );
    fprintf( stderr, "option:  a : add an entry\n" );
    fprintf( stderr, "        r : read an entry\n" );
    fprintf( stderr, "        d : delete an entry\n" );
} /* end showUsage() */

/*****
 * numNamePieces()
 *   Returns the number of pieces in a string name.
 *****/

```

```

*****/
int
numNamePieces(
    char *    string      /* In--String whose pieces are to be counted */
)
{
    int      count;      /* Number of pieces found */
    char *   currSep;    /* Pointer to separator between pieces */

    if( string == NULL ) /* If nothing there, no pieces */
        return( 0 );
    count = 1;          /* Otherwise, there's at least one */

    /*
     * If the first character is a /, it's not really separating
     * two pieces so we want to ignore it here.
     */
    if( *string == '/' )
        currSep = string + 1;
    else
        currSep = string;

    /* How many pieces are there? */
    while( (currSep = strchr( currSep, '/' )) != NULL ) {
        count++;
        currSep++; /* Begin at next character */
    }

    return( count );
} /* end numNamePieces() */

/*****
 * splitNamePiece()
 *   Divides a piece of a name (string) into its XDS attribute type
 *   and value.
 *****/
void
splitNamePiece(
    char *    string, /* In--String to be broken down */
    OM_string * type, /* Out--XDS type of this piece of the name */
    char **   value,  /* Out--Pointer to beginning of the value part */
) /* of string */
{
    char *    equalSign; /* Location of the = within string */

    /*
     * If the string contains an equal sign, this is probably a
     * typed name. Check for all the attribute types allowed by
     * the default schema.
     */
    if( (equalSign = strchr( string, '=' )) != NULL ) {

        *value = equalSign + 1;

        if( ( strcmp( string, "C=", 2 ) == 0 ) ||
            ( strcmp( string, "c=", 2 ) == 0 ) )

```

```

        *type = DS_A_COUNTRY_NAME;

    else if(( strcmp( string, "O=", 2 ) == 0 ) ||
            ( strcmp( string, "o=", 2 ) == 0 ))
        *type = DS_A_ORG_NAME;

    else if(( strcmp( string, "OU=", 3 ) == 0 ) ||
            ( strcmp( string, "ou=", 3 ) == 0 ))
        *type = DS_A_ORG_UNIT_NAME;

    else if(( strcmp( string, "LN=", 3 ) == 0 ) ||
            ( strcmp( string, "ln=", 3 ) == 0 ))
        *type = DS_A_LOCALITY_NAME;

    else if(( strcmp( string, "CN=", 3 ) == 0 ) ||
            ( strcmp( string, "cn=", 3 ) == 0 ))
        *type = DS_A_COMMON_NAME;

    else if(( strcmp( string, "S=", 2 ) == 0 ) ||
            ( strcmp( string, "s=", 2 ) == 0 ))
        *type = DS_A_STATE_OR_PROV_NAME;

    /*
     * If this did not appear to be a type allowed by the
     * default schema, consider the whole string as the
     * value (whose type is "typeless").
     */
    else {
        *type = DSX_TYPELESS_RDN;
        *value = string;
    }
}

/*
 * If the string does not contain an equal sign, this is a
 * typeless name.
 */
else {
    *type = DSX_TYPELESS_RDN;
    *value = string;
}

} /* end splitNamePiece() */

/*****
 * extractValue()
 * Pulls the value of a particular attribute from a private object that
 * was received using ds_read().
 * Returns:
 * OM_SUCCESS      If successful.
 * OM_NO_SUCH_OBJECT  If no values for the attribute
 *                   were found.
 * other           Any value returned by one of the
 *                   om_get() calls.
 *****/
OM_return_code
extractValue(

```

```

OM_private_object object,      /* In--Object to extract from */
OM_string *      attribute,   /* In--Attribute to extract */
char *          value        /* Out--Value found */
)
{
    OM_public_object attrList;
    OM_public_object attrType;
    OM_public_object attrValue;
    OM_public_object entry;
    int i;
    OM_return_code omStatus;
    OM_value_position total;
    OM_value_position totalAttributes;
    OM_type xdsIncludedTypes[] = { 0, /* Place holder */
                                    0 }; /* Null terminator */

    /*
     * Get the entry from the object returned by ds_read().
     */
    xdsIncludedTypes[0] = DS_ENTRY;
    omStatus = om_get( object, /* Object to extract from */
                     OM_EXCLUDE_ALL_BUT_THESE_TYPES+OM_EXCLUDE_SUBOBJECTS,
                     /* Only want what is in */
                     /* xdsIncludedTypes, don't */
                     /* include subobjects */
                     xdsIncludedTypes, /* What to get */
                     OM_TRUE, /* Convert strings to local char set */
                     OM_ALL_VALUES, /* Start with first value */
                     OM_ALL_VALUES, /* End with last value */
                     &entry, /* Put the entry here */
                     &total ); /* Put number of attribute */
                                /* descriptors here */

    if( omStatus != OM_SUCCESS ) {
        fprintf( stderr, "om_get( entry ) returned error %d\n", omStatus );
        return( omStatus );
    }
    if( total <= 0 ) { /* Make sure something was returned */
        fprintf( stderr,
            "Number of descriptors returned by om_get( entry ) was %d\n", total );
        return( OM_NO_SUCH_OBJECT );
    }

    /*
     * Get the attribute list from the entry.
     */
    xdsIncludedTypes[0] = DS_ATTRIBUTES;
    omStatus = om_get( entry->value.object.object,
                     OM_EXCLUDE_ALL_BUT_THESE_TYPES+OM_EXCLUDE_SUBOBJECTS,
                     xdsIncludedTypes, OM_TRUE, OM_ALL_VALUES,
                     OM_ALL_VALUES, &attrList, &totalAttributes );
    if( omStatus != OM_SUCCESS ) {
        fprintf( stderr, "om_get( attrList ) returned error %d\n", omStatus );
        return( omStatus );
    }
    if( totalAttributes <= 0 ) { /* Make sure something was returned */
        fprintf( stderr,
            "Number of descriptors returned by om_get( attrList ) was %d\n",
            total );
    }
}

```

```

return( OM_NO_SUCH_OBJECT );
}

/*
 * Search the list for the attribute with the proper type.
 */
for( i = 0; i < totalAttributes; i++ ) {
    xdsIncludedTypes[0] = DS_ATTRIBUTE_TYPE;
    omStatus = om_get( (attrList+i)->value.object.object,
        OM_EXCLUDE_ALL_BUT_THESE_TYPES+OM_EXCLUDE_SUBOBJECTS,
        xdsIncludedTypes, OM_TRUE, OM_ALL_VALUES,
        OM_ALL_VALUES, &attrType, &total );
    if( omStatus != OM_SUCCESS ) {
        fprintf( stderr, "om_get( attrType ) [i = %d] returned error %d\n",
            i, omStatus );
        return( omStatus );
    }
    if( total <= 0 ) {          /* Make sure something was returned */
        fprintf( stderr,
            "Number of descriptors returned by om_get( attrType ) [i = %d] was %d\n",
            i, total );
        return( OM_NO_SUCH_OBJECT );
    }
    if( strncmp( attrType->value.string.elements, attribute->elements,
        attribute->length ) == 0 )
        break;          /* If we found a match, quit looking. */
}
if( i == totalAttributes ) { /* Verify that we found a match. */
    fprintf( stderr,
        "%s: extractValue() could not find requested attribute\n" );
    return( OM_NOT_PRESENT );
}

/*
 * Get the attribute value from the corresponding item in the
 * attribute list.
 */
xdsIncludedTypes[0] = DS_ATTRIBUTE_VALUES;
omStatus = om_get( (attrList+i)->value.object.object,
    OM_EXCLUDE_ALL_BUT_THESE_TYPES+OM_EXCLUDE_SUBOBJECTS,
    xdsIncludedTypes, OM_TRUE, OM_ALL_VALUES,
    OM_ALL_VALUES, &attrValue, &total );
if( omStatus != OM_SUCCESS ) {
    fprintf( stderr, "om_get( attrValue ) returned error %d\n", omStatus );
    return( omStatus );
}
if( total <= 0 ) {          /* Make sure something was returned */
    fprintf( stderr,
        "Number of descriptors returned by om_get( attrValue ) was %d\n", total );
    return( OM_NO_SUCH_OBJECT );
}

/*
 * Copy the value(s) into the buffer for return to the caller.
 */
for(i=0;;) {
    strncpy( value, attrValue->value.string.elements,
        attrValue->value.string.length );
}

```

```

        value += attrValue->value.string.length;
        if (++i == total)
            break;
        attrValue++; *value++ = ';'; *value++ = ' ';
    }
    *value = '\0';

    /*
     * Free up the resources we don't need any more and return.
     */
    om_delete( attrValue );
    om_delete( attrType );
    om_delete( attrList );
    om_delete( entry );
    return( OM_SUCCESS );
} /* end extractValue() */

/*****
 * stringToXdsName()
 * Converts a string that is a DCE name to an XDS name object (class
 * DS_C_DS_DN). Returns one of the following:
 *     OM_SUCCESS      If successful.
 *     OM_MEMORY_INSUFFICIENT  If a malloc fails.
 *     OM_PERMANENT_ERROR  If the name is not in a valid format.
 *     OM_SYSTEM_ERROR    If the local cell's name cannot be
 *                        determined.
 *
 * Technically, the space obtained here through malloc() needs
 * to be returned to the system when it is no longer needed.
 * If this was a more complex application, this function would
 * probably malloc all the space it needs at once and require
 * calling routines to free the space when finished with it.
 *****/
OM_return_code
stringToXdsName(
    char *      origString, /* In--String name to be converted */
    OM_object * xdsNameObj  /* Out--Pointer to XDS name object */
)
{
    OM_descriptor *  ava;          /* DS_C_AVA object */
    char *          cellName;     /* Name of this cell */
    OM_object       dsdn;         /* DS_C_DS_DN object */
    char *          end;          /* End of name piece */
    int             index;        /* Index into DS_C_DS_DN object */
    int             numberOfPieces; /* Number of pieces in the name */
    unsigned long   rc;           /* Return code for some functions */
    OM_descriptor * rdn;          /* DS_C_RDN object */
    char *          start;        /* Beginning of piece of name */
    char *          string;       /* Copy of origString that
                                   we can use */

    OM_string       type;         /* Type of one piece of the name */
    char *          value;        /* Piece of the name */

    /*
     * A DS_C_AVA object only contains pointers to the strings that
     * represent the pieces of the name, not the contents of the strings

```

```

* themselves. So we'll make a copy of the string passed in to
* guarantee that these pieces survive in case the programmer alters
* or reuses the original string.
*
* In addition, all valid names should begin with one of the
* following symbols:
*   /...      Fully qualified name (from global root). For
*             these, we need to ignore the /...
*   /.:      Partially qualified name (from local cell root).
*             For these, we must replace the /.: with the name
*             of the local cell name
* If we see anything else, we'll return with an error. (Notice that
* /: is a valid DCE name, but refers to the file system's namespace.
* Filenames cannot be accessed through CDS, GDS, or XDS.)
*/
if( strncmp( origString, "/.../" , 5 ) == 0 ) {
    string = (char *)malloc( strlen(origString+5) + 1 );
    if( string == NULL )          /* malloc() failed */
        return OM_MEMORY_INSUFFICIENT;
    strcpy( string, origString+5 );
}
else if( strncmp( origString, "/./", 4 ) == 0 ) {
    dce_cf_get_cell_name( &cellName, &rc );
    if( rc != 0 )              /* Could not get cell name */
        return OM_SYSTEM_ERROR;

    /*
     * The cell name will have /.../ on the front, so we will skip
     * over it as we add it to the string (by always starting at its
     * fifth character).
     */
    string = (char *)malloc( strlen(origString+4) + strlen(cellName+5) + 2 );
    if( string == NULL )          /* malloc() failed */
        return OM_MEMORY_INSUFFICIENT;
    strcpy( string, cellName+5 );
    strcat( string, "/" );
    strcat( string, origString+4 );
}
else
    /* Invalid name format */
    return OM_PERMANENT_ERROR;

/*
 * Count the number of pieces in the name that will have to be dealt with.
 */
numberOfPieces = numNamePieces( string );

/*
 * Allocate memory for the DS_C_DS_DN object. We will need an
 * OM_descriptor for each name piece, one for the class identifier,
 * and one for the null terminator.
 */
dsdn = (OM_object)malloc( (numberOfPieces + 2) * sizeof(OM_descriptor) );
if( dsdn == NULL )              /* malloc() failed */
    return OM_MEMORY_INSUFFICIENT;

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

```

```

*/
FILL_OMD_XOM_STRING( dsdn, 0, OM_CLASS,
                    OM_S_OBJECT_IDENTIFIER_STRING, DS_C_DS_DN )

/*
 * For each piece of string, do the following:
 *   Break off the next piece of the string
 *   Build a DS_C_AVA object to show the type and value
 *   of this piece of the name
 *   Wrap the DS_C_AVA up in a DS_C_RDN object
 *   Add the DS_C_RDN to the DS_C_DS_DN object
 */
for( start=string, index=1 ;
      index <= numberOfPieces ; index++, start=end+1 ) {

/*
 * Find the next delimiter and replace it with a null byte
 * so the piece of the name is effectively separated from
 * the rest of the string.
 */
end = strchr( start, '/' );
if( end != NULL )
    *end = '\0';
else
    /* If this is the last piece, there won't be */
    /* a '/' at the end, just a null byte.      */
    end = strchr( start, '\0' );

/*
 * Allocate space for a DS_C_AVA object and fill in its entries:
 *   DS_C_AVA class identifier
 *   AVA's type
 *   AVA's value
 *   null terminator
 */
ava = (OM_descriptor *)malloc( sizeof(OM_descriptor) * 4 );
if( ava == NULL )
    /* malloc() failed */
    return OM_MEMORY_INSUFFICIENT;
FILL_OMD_XOM_STRING( ava, 0, OM_CLASS,
                    OM_S_OBJECT_IDENTIFIER_STRING, DS_C_AVA )
splitNamePiece( start, &type, &value );
FILL_OMD_XOM_STRING( ava, 1, DS_ATTRIBUTE_TYPE,
                    OM_S_OBJECT_IDENTIFIER_STRING, type )
FILL_OMD_STRING( ava, 2, DS_ATTRIBUTE_VALUES,
                OM_S_PRINTABLE_STRING | OM_S_LOCAL_STRING, value )
FILL_OMD_NULL( ava, 3 )

/*
 * Allocate space for a DS_C_RDN object and fill in its entries:
 *   DS_C_RDN class identifier
 *   AVA it contains
 *   null terminator
 */
rdn = (OM_descriptor *)malloc( sizeof(OM_descriptor) * 3 );
if( rdn == NULL )
    /* malloc() failed */
    return OM_MEMORY_INSUFFICIENT;
FILL_OMD_XOM_STRING( rdn, 0, OM_CLASS, OM_S_OBJECT, DS_C_DS_RDN )
FILL_OMD_STRUCT( rdn, 1, DS_AVAS, OM_S_OBJECT, ava )
FILL_OMD_NULL( rdn, 2 )

```

```

    /* Add the DS_C_RDN object to the DS_C_DS_DN object. */
    FILL_OMD_STRUCT( dsdn, index, DS_RDNS, OM_S_OBJECT, rdn )
}

/*
 * Null terminate the DS_C_DS_DN, tell the calling routine
 * where to find it, and return.
 */
FILL_OMD_NULL( dsdn, index )
*xdsNameObj = dsdn;
return( OM_SUCCESS );

} /* end stringToXdsName() */

/*****
 * handleDSError()
 *   Extracts the error number from a DS_status return code, prints it
 *   in an error message, then terminates the program.
 *****/
void
handleDSError(
    char *      header,      /* In--Name of function whose return code */
                                /* is being checked */
    DS_status   returnCode /* In--Return code to be checked */
)
{
    OM_type      includeDSProblem[] = { DS_PROBLEM, 0 };
    OM_return_code omStatus;
    OM_public_object problem;
    OM_value_position total;

    /*
     * A DS_status return code is an object. It will be one of the
     * subclasses of the class DS_C_ERROR. What we want from it is
     * the value of the attribute DS_PROBLEM.
     */
    omStatus = om_get( returnCode,
                      OM_EXCLUDE_ALL_BUT_THESE_TYPES+OM_EXCLUDE_SUBOBJECTS,
                      includeDSProblem,
                      OM_TRUE,
                      OM_ALL_VALUES,
                      OM_ALL_VALUES,
                      &problem,
                      &total );

    /*
     * Make sure we successfully extracted the problem number and print
     * the error message before quitting.
     */
    if( (omStatus == OM_SUCCESS) && (total > 0) )
        printf( "%s returned error %d\n", header, problem->value.enumeration );
    else
        printf( "%s failed for unknown reason\n", header );

    exit( 1 );
}

```

```

/*****
 * Main program
 */
void
main(
    int      argc,
    char *   argv[]
)
{
    DS_status      dsStatus;
    OM_sint        invokeID;
    char           newName[MAX_NAME_LEN];
    char           newPhoneNum[MAX_PHONE_LEN];
    char           newSurname[MAX_SURNAME_LEN];
    OM_return_code omStatus;
    char           phoneNumRead[MAX_PHONE_LEN];
    int            rc = 0;
    OM_private_object readResult;
    OM_private_object session;
    char           surnameRead[MAX_SURNAME_LEN];
    OM_object      xdsName;
    OM_workspace   xdsWorkspace;
    char           operation;

    /* STEP 1
     *
     * Examine command-line argument.
     */
    operation = *argv[1];
    if ( (operation != 'r') && (operation != 'a') && (operation != 'd') ) {
        showUsage( argv[0] );
        exit( 1 );
    }

    /* STEP 2
     *
     * Initialize the XDS workspace.
     */
    xdsWorkspace = ds_initialize( );
    if( xdsWorkspace == NULL ) {
        fprintf( stderr, "ds_initialize() failed\n" );
        exit( 1 );
    }

    /* STEP 3
     *
     * Pull in the packages that contain the XDS features we need.
     */
    dsStatus = ds_version( featureList, xdsWorkspace );
    if( dsStatus != DS_SUCCESS )
        handleDSError( "ds_version()", dsStatus );

    /* STEP 4
     *
     * Find out what name the user wants to use in the namespace and

```

```

* convert it to and XDS object. We do this conversion dynamically
* (not using static structures defined at the top of the program)
* because we don't know how long the name will be.
*/
switch( operation ) {
case 'r' :
    printf( "What name do you want to read? " );
    break;
case 'a' :
    printf( "What name do you want to add? " );
    break;
case 'd' :
    printf( "What name do you want to delete? " );
    break;
}

/* STEP 5 */

gets( newName );
omStatus = stringToXdsName( newName, &xdsName );
if( omStatus != OM_SUCCESS ) {
    fprintf( stderr,
            "stringToXdsName() failed with OM error %d\n", omStatus );
    exit( 1 );
}

if ( operation == 'a' ) { /* add operation requires additional input */
    /*
    * Get the person's real name from the user and place it in the
    * XDS object already defined at the top of the program (xdsSurname).
    * We are requiring a name, so we will loop until we get one.
    */
    do {
        printf( "What is this person's surname? " );
        gets( newSurname );
    } while ( *newSurname == '\0' );
    FILL_OMD_STRING( xdsSurname, 2, DS_ATTRIBUTE_VALUES,
                    OM_S_TELETEX_STRING | OM_S_LOCAL_STRING, newSurname )

    /*
    * Get the person's phone number from the user and place it in the
    * XDS object already defined at the top of the program (xdsPhoneNum).
    * A phone number is not required, so if none is given we will use
    * the default already stored in the structure.
    */
    printf( "What is this person's phone number? " );
    gets( newPhoneNum );
    if( *newPhoneNum != '\0' ) {
        FILL_OMD_STRING( xdsPhoneNum, 2, DS_ATTRIBUTE_VALUES,
                        OM_S_PRINTABLE_STRING | OM_S_LOCAL_STRING, newPhoneNum )
    }
}

/* STEP 6
*
* Open the session with the namespace:
* bind (without credentials) to the default server.
*/

```

```

dsStatus = ds_bind( DS_DEFAULT_SESSION, xdsWorkspace, &session );
if( dsStatus != DS_SUCCESS )
    handleDSError( "ds_bind()", dsStatus );

/* STEP 7 */

switch( operation ) { /* perform the requested operation */

/*
 * Add entry to the namespace. The xdsSurname and xdsPhoneNum
 * objects are already contained within an attribute list object
 * (xdsAttributesToAdd).
 */
case 'a' :
    dsStatus = ds_add_entry( session, DS_DEFAULT_CONTEXT, xdsName,
                            xdsAttributesToAdd, &invokeID );
    if( dsStatus != DS_SUCCESS )
        handleDSError( "ds_add_entry()", dsStatus );

    break;
/* FALL THROUGH */

/*
 * Read the entry of the name supplied.
 */
case 'r' :
    dsStatus = ds_read( session, DS_DEFAULT_CONTEXT, xdsName,
                       xdsAttributeSelection, &readResult, &invokeID );
    if( dsStatus != DS_SUCCESS )
        handleDSError( "ds_read()", dsStatus );

/*
 * Get each attribute from the object read and print them.
 */
omStatus = extractValue( readResult, &DS_A_SURNAME, surnameRead );
if( omStatus != OM_SUCCESS ) {
    printf( "** Surname could not be read\n" );
    strcpy( surnameRead, "(unknown)" );
    rc = 1;
}
omStatus = extractValue( readResult, &DS_A_PHONE_NBR, phoneNumRead );
if( omStatus != OM_SUCCESS ) {
    printf( "** Phone number could not be read\n" );
    strcpy( phoneNumRead, "(unknown)" );
    rc = 1;
}
printf( "The phone number for %s is %s.\n",
        surnameRead, phoneNumRead );

    break;

/*
 * delete the entry from the namespace.
 */
case 'd' :
    dsStatus = ds_remove_entry( session, DS_DEFAULT_CONTEXT, xdsName,
                               &invokeID );

```

```

    if( dsStatus != DS_SUCCESS )
        handleDSError( "ds_remove_entry()", dsStatus );
    else
        printf( "The entry has been deleted.\n" );
        break;
}

/*
 * Clean up and exit.
 */
/* STEP 8 */
dsStatus = ds_unbind( session );
if( dsStatus != DS_SUCCESS )
    handleDSError( "ds_unbind()", dsStatus );

/* STEP 9 */
dsStatus = ds_shutdown( xdsWorkspace );
if( dsStatus != DS_SUCCESS )
    handleDSError( "ds_shutdown()", dsStatus );

exit( rc );
} /* end main() */

```

Chapter 8. Using Threads with the XDS/XOM API

Some programs work well when they are structured as multiple flows of control. Other programs may show better performance when they are multithreaded allowing the multiple threads to be mapped to multiple processors when they are available.

XDS/XOM supports multithreaded applications. Writing multithreaded applications over XDS/XOM imposes new requirements on programmers: they must manage the threads, synchronize threads' access to global resources, and make choices about thread scheduling and priorities.

This chapter describes a simple XDS/XOM application that uses threads. (Refer to the *z/OS DCE Application Development Guide: Core Components* for more information on DCE threads.)

The XDS/XOM API calls do not change when an application makes use of DCE threads. The service underneath XDS/XOM API is designed to be:

- Thread-safe, to allow multiple threads to safely access shared data
- Cancel-safe, to handle unexpected cancelation of a thread in an application program

Figure 47 shows an example of how an application can issue XDS/XOM calls from within different threads.

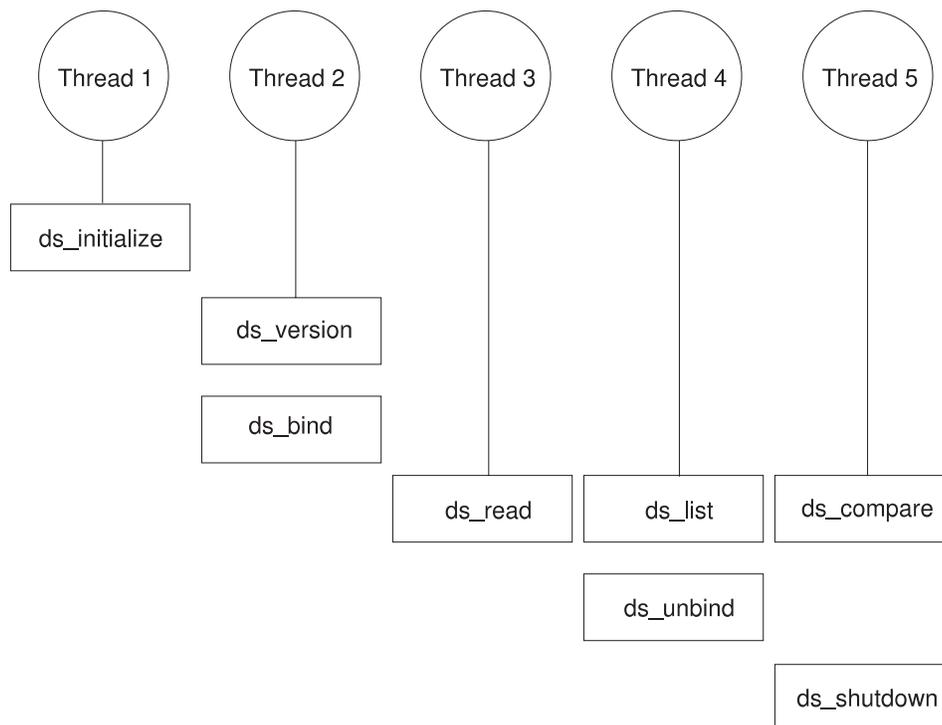


Figure 47. Issuing XDS/XOM Calls from within Different Threads

The order of thread completion is not defined; however, XDS/XOM has an inherent ordering. Multithreaded XDS applications must adhere to the following order of execution:

1. **ds_initialize()**
2. **ds_version()** (optional)

3. **ds_bind()**
4. other XDS calls in sequence or parallel from multiple threads
5. **ds_unbind()**
6. **ds_shutdown()**

Multithreaded XOM applications must adhere to the following order of execution:

1. **ds_initialize()**
2. XOM calls in sequence or parallel from multiple threads
3. **ds_shutdown()**

The XDS/XOM API will return an appropriate error code if these sequences are not adhered to. For example the following errors are returned:

DS_E_BUSY	If ds_unbind() is called while there are still outstanding operations, or if ds_shutdown() is called before all directory connections have been released by ds_unbind() .
OM_NO_WORKSPACE	If any XOM API calls are made before calling ds_initialize() , or if a call to ds_shutdown() completes while there are outstanding XOM operations on the same workspace. In the latter case, these XOM operations will not be performed.

Overview of Example Threads Program

The example program is called **thradd**. The **thradd** program is a multithreaded XDS application that adds CDS object entries to a CDS directory. Each thread performs a **ds_add_entry()** call. The information for each entry to be added is read from an input file.

The **thradd** program can also be used to reset the directory to its original state. This is achieved by invoking **thradd** with a **-d** command-line argument. In this case, **thradd** uses the same input file and calls **ds_remove_entry()** for each entry. The **ds_remove_entry()** calls are also done in separate threads.

To keep the program short and clear, it works with a fixed tree for the upper nodes (like **example.c**). Static descriptors for both an X.500 and a DNS-type cell name are provided. The **CELLNAME_TYPE** pre-processor directive is used to control which name format is in effect in the DCE cell. This fixed upper tree is assumed to exist prior to running **thradd**. The input file contains the common name, the surname, and the phone number of each **Organizational-Person** entry to be added.

For simplicity, only **pthread_join()** is used for synchronization purposes; mutexes are not used.

The **thradd** program could be enhanced to satisfy the following scenarios:

- As a server program for interactive directory actions from different users. The **thradd** program simulates a server program which gets requests from different users to add entries to a directory. In the case of **thradd**, the users' interactive input is simulated through the entries in the input file. Each line of input represents a different directory entry, and **thradd** uses a separate thread for each line.
- Initialization of the directory with data from file. The **thradd** program could be enhanced to read generic attribute information for a variety of directory object classes from a file, and to add the corresponding entries to the directory.

User Interface

The **thradd** program is called from the command line as follows:

thradd [-d] [-f *file name*]

- d** If the option **-d** is set, the entries in the file that are found in the CDS directory used to create the entries are deleted. If not specified, they are added.
- f *file name*** The option **-f** specifies the name of the input file. If no input file is specified, then a default file name of **thradd.dat** is used.

Input File Format

The input file can contain any number of lines. Each line represents a directory entry of an organizational person. Each line must contain the following three attributes for each entry:

<common name> *<surname>* *<phone number>*

The attributes must be strings without space characters. Lines containing less than three strings are rejected by the program; any input on a line after the first three strings is ignored and can be used for comments. The attributes are separated by one or more space characters.

The input strings are not verified for their relevant attribute syntax. A wrong attribute syntax will result in either a **ds_add_entry()** error or a **ds_remove_entry()** error.

The following would be a valid input file for **thradd**:

Anna	Meister	010101
Erwin	Reiter	020202
Gerhard	Schulz	030303
Gottfried	Schmid	040404
Heidrun	Blum	050505
Hermann	Meier	060606
Josefa	Fischer	070707
Jutta	Arndt	080808
Leopold	Huber	090909
Magdalena	Schuster	101010
Margot	Junge	111111

Program Output

The **thradd** program writes messages to *stdout* for every action done by a thread. The order of the output can differ from the order in the input file; it depends on the execution of the different threads.

Errors are reported to *stderr*.

Prerequisites

The directory must be active before running **thradd**. The **thradd** program should always be invoked twice with the same input file: first without and then with option **-d**. This guarantees that the directory is reset to its original state.

Description of Thradd Example Program

The **thradd** program has a similar structure to the example XDS programs in the previous chapter. Therefore, only a short general outline of the program is given here. The thread specifics are described in detail in the next section.

The static descriptors for the fixed tree (depending on the **CELLNAME_TYPE** pre-processor directive.) are declared in the **thradd.h** header file. Listings of both the **thradd.c** application and the **thradd.h** header file are included in later sections of this chapter.

The main routine scans the command-line options, initializes the XDS workspace and, if working in adding mode, binds to the directory without credentials, adds the fixed tree of upper nodes, and then unbinds from the directory.

Each line of the input file is processed in turn by a *while* loop (until the end of file is reached). The *while* loop contains two *for* loops. The first *for* loop creates a separate thread for each line of the input file, up to a maximum of **MAX_THREAD_NO** of threads.

The **add_or_remove()** procedure, which adds an entry to or removes an entry from the directory, is the starting point of each thread's processing.

The second *for* loop waits for termination of the threads and then releases the resources used by the threads.

When the entire input file has been processed, **thradd** closes the connection to the directory server.

Finally, the XDS workspace is closed.

Figure 48 on page 195 shows the program flow.

The source code for this program can be found in the **/usr/lpp/dce/examples/xdsxom/thradd** directory.

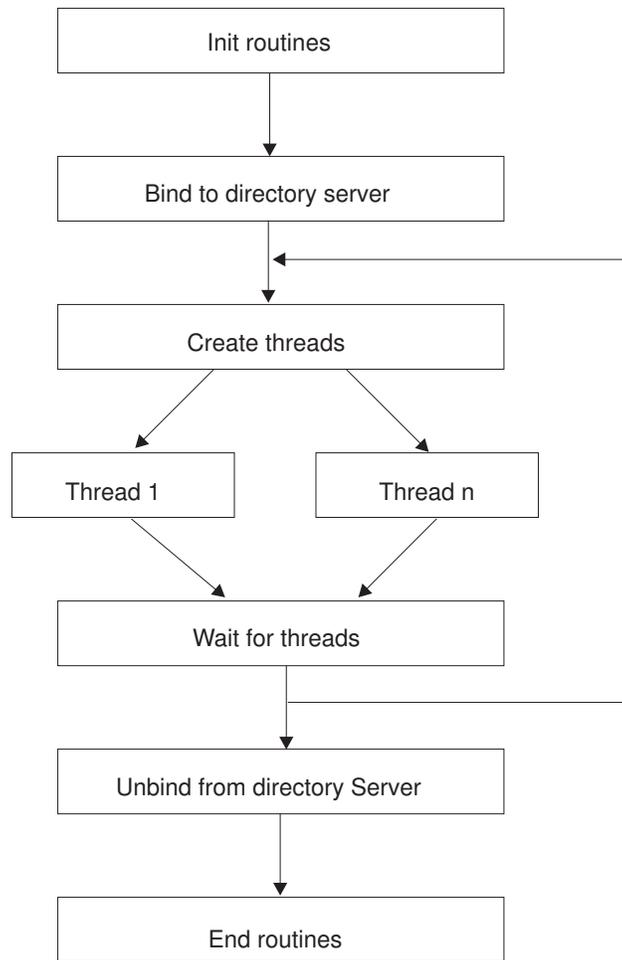


Figure 48. Program Flow for the thradd Sample Program

Detailed Description of Thread Specifics

The program consists of the following general steps:

1. Include the header file **pthread.h**.
2. Define a parameter block structure type for the thread start routine, **add_or_remove**.
3. Declare arrays for thread handles and parameter blocks.
4. Read the input file line by line.
5. Update the parameter block.
6. Create the thread.
7. Wait for the termination of the thread.
8. Release the resources used by the thread.
9. Define the thread start routine.
10. Declare local variables needed for descriptors for the objects read from the input file.

The following paragraphs describe the corresponding step numbers from the program listing in the next section:

Step 1 includes the header file **pthread.h** which is required for thread programming.

Step 2 defines a parameter block structure type for the thread start routine. A thread start routine must have exactly one parameter. However, **add_or_remove()** requires three parameters (session object, input line and operating mode). The structure **pb_add_or_remove** is defined as the parameter block for these components. Therefore, the single parameter block contains the three parameters required by **add_or_remove()**.

Step 3 declares arrays for thread handles and parameter blocks. The routine which creates the thread (*main* in this case) must maintain the following information for each thread:

- A thread handle of type **pthread_t** to identify the thread for join and detach calls.
- A thread specific parameter block that cannot be accessed by any other thread. This makes sure that a parameter for one thread is not overwritten by another thread.

Step 4 reads the input file line by line. A thread is created for each line. A maximum **MAX_THREAD_NO** of threads are created in parallel. The program then waits for the termination of the created threads so that it can release the resources used by these threads, allowing it to create new threads for remaining input lines (if any).

The absolute maximum number of threads working in parallel depends on system limits; for **thradd** a value of 10 was chosen for **MAX_THREAD_NO** (see **thradd.h**), which is well below the maximum on most systems.

Step 5 updates the parameter block. For each thread a different element of the array of parameter blocks is used.

Step 6 creates the thread. The thread is created by using the function **pthread_create()**. The function has four parameters:

- The thread handle (output) is stored in an element of the *threads* array which is of type **pthread_t**.
- For the thread characteristics, the default **pthread_attr_default** is used.
- The start routine for this thread is **add_or_remove()**.
- The parameter passed to **add_or_remove()** is a pointer to an element of the array of parameter blocks, one of the elements of the (*param_block*) array.

Step 7 waits for the termination of the thread. The **pthread_join()** routine is called with the thread handle as the input parameter. The program waits for the termination of the thread. If the thread has already terminated, then **pthread_join()** returns immediately. The second parameter of **pthread_join()** contains the return value of the start function; here it is a dummy value because **add_or_remove()** returns a *void*. **add_or_remove()** is designed as a *void* function because the calling routine does not have to deal with error cases. The **add_or_remove()** routine prints status messages itself to show the processing order of the threads. Normally a status should be returned to the application.

Step 8 releases the resources used by the thread. The thread handle is used as input for the function **pthread_detach()**, which releases the resources (for example, memory) used by the thread.

Step 9 defines the thread start routine. As previously mentioned, the thread start routine must have exactly one parameter. In this case, it is a pointer to the parameter block structure defined in **Step 3**.

Step 10 declares local variables needed for descriptors for the objects read from the input file. These descriptors are variables and are declared as automatic because of the reentrancy requirement. In the previous example program, descriptors were declared static. For this example, this is only possible for the constant descriptors declared in **thradd.h**.

This example shows only a small part of the possibilities of multithreaded XDS programming.

The thradd.c Code

The following code is a listing of the **thradd.c** program:

```
/*
 * The program operates in two modes; it adds or removes entries of
 * object type organizational person to/from a directory. The information
 * about the entries is read from a file.
 *
 * The program requires that there exists a tree in the directory.
 * This must be set up prior to running the program. Consult the
 * README file for more information.
 *
 * Information about the organizational persons to be added or removed is
 * read from the input file. It may contain any number of lines, where
 * each line must have the following syntax:
 *
 *     <common name> <surname> <phone number>
 *     Each item must be a string without a blank.
 *
 * Lines containing less than 3 strings are rejected by the program.
 * The program does not check to see if the strings conform to the
 * appropriate attribute syntax; i.e. a wrong attribute syntax will
 * lead to a ds_add_entry error, or to a ds_remove_entry error.
 *
 * Usage: thradd [-d] [-f<file name>]
 *   -d           If the option -d is set, the entries in the file and the
 *                tree described above are removed, otherwise they are added.
 *   -f<file name> The option -f specifies the name of the input file.
 *                If left out, the default "thradd.dat" is used.
 */

/* Step 1 */

/*
 * Header file for thread programming:
 */
#include <pthread.h>
#include <dce/euvsplws.h>

#include <stdio.h>
#include <xom.h>
#include <xds.h>
#include <xdsbdcp.h>
#include <xdsrgds.h>
#include <xdsrds.h>
#include "thradd.h"           /* static data structures. */
```

```

/* Step 2 */

/*
 * typedef for parameter block of function add_or_remove
 * (this is necessary because start functions of a thread
 * takes only 1 parameter). The following 3 parameters are
 * passed to add_or_remove:
 *
 * Input - Session object from the ds_bind call
 * Input - Buffer with the entry information
 * Input - "adding" or "removing" mode?
 */
typedef struct {
    OM_private_object session;
    char          line[MAX_LINE_LEN+1];
    int           do_remove;
} pb_add_or_remove;

/*
 * static constants:
 *
 * Default name for input file containing entry information.
 */
static char fn_default[] = "thradd.dat";

/*
 * function declarations:
 */
char *own_fgets(char *s, int n, FILE *f);
void add_or_remove(pb_add_or_remove *pb);

int
main(
    int argc,
    char *argv[]
)
{
    OM_workspace      workspace;          /* workspace for objects          */
    OM_private_object bound_session;      /* Holds the Session object which */
                                          /* is returned by ds_bind()      */
    FILE              *fp;                /* file pointer for input file    */
    int               do_remove = FALSE; /* option -d for remove set ?    */
    int               error      = FALSE; /* error in options ?            */
    int               is_eof     = FALSE; /* EOF in input file reached ?   */
    int               thread_count; /* actual number of created threads */
    char              *file_name; /* pointer to input file name    */

    /* Step 3 */

    pthread_t        threads[MAX_THREAD_NO]; /* thread table          */
    pb_add_or_remove param_block[MAX_THREAD_NO]; /* 1 param block for start */
                                          /* routine per thread     */

    int              dummy;

```

```

int          c;
int          i;

/*
 * scan options -d and -f
 */
file_name = fn_default;

i=1;
while ( ( i < argc ) && ( error == FALSE ) )
{
    if ( argv[i][0] == '-' )
    {
        switch ( argv[i][1] )
        {
            case 'd':
                do_remove = TRUE;
                break;
            case 'f':
                file_name = &argv[i][2];
                break;
            default:
                error = TRUE;
                break;
        }
    }
    else
    {
        error = TRUE;
    }
    i++;
}

if ( error )
{
    printf("usage: %s [-d] [-f<file name>]\n", argv[0]);
    return(FAILURE);
}

if ( ( fp = fopen(file_name, "r") ) == (FILE *) NULL )
{
    printf("fopen() error, file name: %s\n", file_name);
    return(FAILURE);
}

/*
 * Initialize a directory workspace for use by XOM.
 */
if ( (workspace = ds_initialize()) == (OM_workspace)0 )
    printf("ds_initialize() error\n");

/*
 * Negotiate the use of the BDCP and GDS packages.
 */

```

```

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

/*
 * Bind to the default GDS server.
 * The returned session object is stored in the private object variable
 * bound_session and is used for further XDS function calls.
 */
if (ds_bind(DS_DEFAULT_SESSION, workspace, &bound_session) != DS_SUCCESS)
    printf("ds_bind() error\n");

/* Step 4 */

/*
 * Add or remove entries described in input file.
 * This is done in parallel, creating up to MAX_THREAD_NO threads
 * at a time.
 */
while (!is_eof)
{
    for (thread_count=0; thread_count<MAX_THREAD_NO; thread_count++)
    {
        /* Step 5 */

        /*
         * Prepare parameter block:
         */
        is_eof = (own_fgets(param_block[thread_count].line,
                           MAX_LINE_LEN, fp) == NULL);
        if (is_eof)
            break;

        param_block[thread_count].session = bound_session;
        param_block[thread_count].do_remove = do_remove;

        /* Step 6 */

        /*
         * Create thread with start routine add_or_remove:
         */
        if (pthread_create(&threads[thread_count], pthread_attr_default,
                           (pthread_startroutine_t) add_or_remove,
                           (pthread_addr_t) &param_block[thread_count])
            != SUCCESS)
            printf("pthread_create() error\n");
    } /* end for */

    /*
     * Wait for termination of the created threads and release resources:
     */
    for (i=0; i<thread_count; i++)
    {

        /* Step 7 */

        /*

```

```

    * Wait for termination of thread
    * (If thread has terminated already, the function returns
    * immediately):
    */
    if (pthread_join(threads[i], (pthread_addr_t) &dummy) != SUCCESS)
        printf("pthread_join() error\n");

    /* Step 8 */

    /*
    * Release resources used by the thread:
    */
    if (pthread_detach(&threads[i]) != SUCCESS)
        printf("pthread_detach() error\n");
} /* end for */
} /* end while */

/*
* Close the connection to the GDS server.
*/
if (ds_unbind(bound_session) != DS_SUCCESS)
    printf("ds_unbind() error\n");

if (om_delete(bound_session) != OM_SUCCESS)
    printf("om_delete() error\n");

/*
* Close the directory workspace.
*/
if (ds_shutdown(workspace) != DS_SUCCESS)
    printf("ds_shutdown() error\n");

fclose(fp);
return(SUCCESS);
} /* end main() */

/* Step 9 */

/*
* Handle (add or remove) a directory entry
*/
void
add_or_remove(
    pb_add_or_remove *pb /* see typedef for parameter information */
)
{
    /*
    * further local variables:
    */
    char common_name[MAX_AT_LEN+1];
    char phone_num[MAX_AT_LEN+1];
    char surname[MAX_AT_LEN+1];
    OM_sint invoke_id;

```

```

/* Step 10 */

/*
 * local variables for descriptors for objects read from file
 */
OM_descriptor   ava_genop[] = {
    OM_OID_DESC(OM_CLASS, DS_C_AVA),
    OM_OID_DESC(DS_ATTRIBUTE_TYPE, DSX_TYPELESS_RDN),
    OM_NULL_DESCRIPTOR, /* place holder */
    OM_NULL_DESCRIPTOR
};

OM_descriptor   rdn_genop[] = {
    OM_OID_DESC(OM_CLASS, DS_C_DS_RDN),
    OM_NULL_DESCRIPTOR, /* place holder */
    OM_NULL_DESCRIPTOR
};

#if CELLNAME_TYPE == DNS_TYPE

OM_descriptor dn_genop[] = {
    OM_OID_DESC(OM_CLASS, DS_C_DS_DN),
    {DS_RDNS, OM_S_OBJECT, {0, rdn1}},
    {DS_RDNS, OM_S_OBJECT, {0, rdn2}},
    OM_NULL_DESCRIPTOR, /* place holder */
    OM_NULL_DESCRIPTOR
};
int obj_name_index=3;

#else

OM_descriptor dn_genop[] = {
    OM_OID_DESC(OM_CLASS, DS_C_DS_DN),
    {DS_RDNS, OM_S_OBJECT, {0, rdn1}},
    {DS_RDNS, OM_S_OBJECT, {0, rdn2}},
    {DS_RDNS, OM_S_OBJECT, {0, rdn3}},
    {DS_RDNS, OM_S_OBJECT, {0, rdn4}},
    OM_NULL_DESCRIPTOR, /* place holder */
    OM_NULL_DESCRIPTOR
};
int obj_name_index=5;

#endif

OM_descriptor   att_phone_num[] = {
    OM_OID_DESC(OM_CLASS, DS_C_ATTRIBUTE),
    OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_PHONE_NBR),
    OM_NULL_DESCRIPTOR, /* place holder */
    OM_NULL_DESCRIPTOR
};

OM_descriptor   att_surname[] = {
    OM_OID_DESC(OM_CLASS, DS_C_ATTRIBUTE),
    OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_SURNAME),
    OM_NULL_DESCRIPTOR, /* place holder */
    OM_NULL_DESCRIPTOR
};

```

```

OM_descriptor  alist_OP[] = {
    OM_OID_DESC(OM_CLASS, DS_C_ATTRIBUTE_LIST),
    {DS_ATTRIBUTES, OM_S_OBJECT, {0, obj_class_OP} },
    OM_NULL_DESCRIPTOR,          /* place holder */
    OM_NULL_DESCRIPTOR,          /* place holder */
    OM_NULL_DESCRIPTOR
};

rdn_genop[1].type = DS_AVAS;
rdn_genop[1].syntax = OM_S_OBJECT;
rdn_genop[1].value.object.padding = 0;
rdn_genop[1].value.object.object = ava_genop;

dn_genop[obj_name_index].type = DS_RDNS;
dn_genop[obj_name_index].syntax = OM_S_OBJECT;
dn_genop[obj_name_index].value.object.padding = 0;
dn_genop[obj_name_index].value.object.object = rdn_genop;

alist_OP[2].type = DS_ATTRIBUTES;
alist_OP[2].syntax = OM_S_OBJECT;
alist_OP[2].value.object.padding = 0;
alist_OP[2].value.object.object = att_surname;

alist_OP[3].type = DS_ATTRIBUTES;
alist_OP[3].syntax = OM_S_OBJECT;
alist_OP[3].value.object.padding = 0;
alist_OP[3].value.object.object = att_phone_num;

if (sscanf(pb->line, "%s %s %s", common_name, surname, phone_num) != 3)
{
    printf("invalid input line: >%s<\n", pb->line);
    return;
}
/*
 * Fill descriptor for common name
 */
ava_genop[2].type = DS_ATTRIBUTE_VALUES;
ava_genop[2].syntax = OM_S_PRINTABLE_STRING | OM_S_LOCAL_STRING;
ava_genop[2].value.string.length = (OM_string_length)strlen(common_name);
ava_genop[2].value.string.elements = common_name;

if (!pb->do_remove)    /* add */
{
    /*
     * Fill descriptors for surname and phone number
     */
    att_surname[2].type = DS_ATTRIBUTE_VALUES;
    att_surname[2].syntax = OM_S_TELETEX_STRING | OM_S_LOCAL_STRING;
    att_surname[2].value.string.length =
        (OM_string_length)strlen(surname);
    att_surname[2].value.string.elements = surname;

    att_phone_num[2].type = DS_ATTRIBUTE_VALUES;
    att_phone_num[2].syntax = OM_S_PRINTABLE_STRING | OM_S_LOCAL_STRING;
    att_phone_num[2].value.string.length =
        (OM_string_length)strlen(phone_num);
    att_phone_num[2].value.string.elements = phone_num;
}

```

```

/*
 * add entry
 */
if (ds_add_entry(pb->session, DS_DEFAULT_CONTEXT, dn_genop,
                alist_OP, &invoke_id) != DS_SUCCESS)
    printf("ds_add_entry() error: %s %s %s\n", common_name,
          surname, phone_num);
else
    printf("entry added: %s %s %s\n", common_name, surname, phone_num);
}
else /* remove */
{
    /*
     * remove entry
     */
    if (ds_remove_entry(pb->session, DS_DEFAULT_CONTEXT, dn_genop,
                       &invoke_id) != DS_SUCCESS)
        printf("ds_remove_entry() error: %s\n", common_name);
    else
        printf("entry removed: %s\n", common_name);
} /* end if */
} /* end add_or_remove() */

```

```

/*
 * read one line with fgets and overwrite new line by a null character
 */

```

```

char *
own_fgets(
    char *s, /* OUT--string read */
    int n, /* IN---maximum number of chars to be read */
    FILE *f /* IN---input file */
)
{
    char *result;
    int i = 0;

    result = fgets(s, n, f);
    if (result != NULL)
    {
        i = strlen(s);
        if (s[i-1] == '\n')
            s[i-1] = '\0';
    }
    return (result);
}

```

The thradd.h Header File

The following code is a listing of the **thradd.h** header file:

```
#ifndef THRADD_H
#define THRADD_H

#ifndef TRUE
#define TRUE (1)
#endif

#ifndef FALSE
#define FALSE (0)
#endif

#define SUCCESS 0
#define FAILURE 1
#define MAX_LINE_LEN 100 /* max length of line in input file */
#define MAX_AT_LEN 100 /* max length of an attribute value */
#define MAX_THREAD_NO 10 /* max number of threads created */

/*
 *      Default this example to use a DNS style cell name.
 */

#define DNS_TYPE 1
#define GDS_TYPE 2
#ifndef CELLNAME_TYPE
#define CELLNAME_TYPE DNS_TYPE
#endif

#if CELLNAME_TYPE == DNS_TYPE
#ifndef DNS_CELLNAME
#define DNS_CELLNAME "cellname"
#endif
#else
#ifndef GDS_CELLNAME_C
#define GDS_CELLNAME_C "US"
#endif
#ifndef GDS_CELLNAME_O
#define GDS_CELLNAME_O "Acme Pepper Co"
#endif
#ifndef GDS_CELLNAME_OU
#define GDS_CELLNAME_OU "Research"
#endif
#endif

#ifndef CDS_DIR_NAME
#define CDS_DIR_NAME "PhoneBook"
#endif

/* The application must export the object identifiers it requires. */

OM_EXPORT (DS_C_AVA)
OM_EXPORT (DS_C_DS_RDN)
OM_EXPORT (DS_C_DS_DN)
```

```

OM_EXPORT (DS_C_ATTRIBUTE)
OM_EXPORT (DS_C_ATTRIBUTE_LIST)

#if CELLNAME_TYPE == GDS_TYPE
    OM_EXPORT(DS_A_COUNTRY_NAME)
    OM_EXPORT(DS_A_ORG_NAME)
    OM_EXPORT(DS_A_ORG_UNIT_NAME)
#endif
OM_EXPORT (DSX_TYPELESS_RDN)
OM_EXPORT (DS_A_OBJECT_CLASS)
OM_EXPORT (DS_A_PHONE_NBR)
OM_EXPORT (DS_A_SURNAME)

OM_EXPORT (DS_O_TOP)
OM_EXPORT (DS_O_PERSON)
OM_EXPORT (DS_O_ORG_PERSON)

static OM_descriptor cds_dir_name[] = {
    OM_OID_DESC(OM_CLASS, DS_C_AVA),
    OM_OID_DESC(DS_ATTRIBUTE_TYPE, DSX_TYPELESS_RDN),
    { DS_ATTRIBUTE_VALUES, OM_S_TELETEX_STRING | OM_S_LOCAL_STRING,
      OM_STRING(CDS_DIR_NAME) },
    OM_NULL_DESCRIPTOR
};

#if CELLNAME_TYPE == DNS_TYPE

    static OM_descriptor cellname[] = {
        OM_OID_DESC(OM_CLASS, DS_C_AVA),
        OM_OID_DESC(DS_ATTRIBUTE_TYPE, DSX_TYPELESS_RDN),
        { DS_ATTRIBUTE_VALUES, OM_S_PRINTABLE_STRING | OM_S_LOCAL_STRING,
          OM_STRING(DNS_CELLNAME) },
        OM_NULL_DESCRIPTOR
    };

    static OM_descriptor rdn1[] = {
        OM_OID_DESC(OM_CLASS, DS_C_DS_RDN),
        { DS_AVAS, OM_S_OBJECT, { 0, cellname } },
        OM_NULL_DESCRIPTOR
    };

    static OM_descriptor rdn2[] = {
        OM_OID_DESC(OM_CLASS, DS_C_DS_RDN),
        { DS_AVAS, OM_S_OBJECT, { 0, cds_dir_name } },
        OM_NULL_DESCRIPTOR
    };

#else

    static OM_descriptor  ava_C[] = {
        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(GDS_CELLNAME_C)},
        OM_NULL_DESCRIPTOR
    };

    static OM_descriptor  ava_O[] = {
        OM_OID_DESC(OM_CLASS, DS_C_AVA),
        OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_ORG_NAME),
        {DS_ATTRIBUTE_VALUES, OM_S_TELETEX_STRING, OM_STRING(GDS_CELLNAME_U)},

```

```

OM_NULL_DESCRIPTOR
};
static OM_descriptor   ava_OU[] = {
OM_OID_DESC(OM_CLASS, DS_C_AVA),
OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_ORG_UNIT_NAME),
{DS_ATTRIBUTE_VALUES, OM_S_TELETEX_STRING, OM_STRING(GDS_CELLNAME_OU)},
OM_NULL_DESCRIPTOR
};

static OM_descriptor   rdn1[] = {
OM_OID_DESC(OM_CLASS, DS_C_DS_RDN),
{DS_AVAS, OM_S_OBJECT, {0, ava_C}},
OM_NULL_DESCRIPTOR
};
static OM_descriptor   rdn2[] = {
OM_OID_DESC(OM_CLASS, DS_C_DS_RDN),
{DS_AVAS, OM_S_OBJECT, {0, ava_0}},
OM_NULL_DESCRIPTOR
};
static OM_descriptor   rdn3[] = {
OM_OID_DESC(OM_CLASS, DS_C_DS_RDN),
{DS_AVAS, OM_S_OBJECT, {0, ava_OU}},
OM_NULL_DESCRIPTOR
};
static OM_descriptor   rdn4[] = {
OM_OID_DESC(OM_CLASS, DS_C_DS_RDN),
{DS_AVAS, OM_S_OBJECT, {0, cds_dir_name}},
OM_NULL_DESCRIPTOR
};

#endif

/* Build up an array of object identifiers for the attributes to be      */
/* added to the directory.                                             */

static OM_descriptor   obj_class_OP[] = {
OM_OID_DESC(OM_CLASS, DS_C_ATTRIBUTE),
OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_OBJECT_CLASS),
OM_OID_DESC(DS_ATTRIBUTE_VALUES, DS_O_TOP),
OM_OID_DESC(DS_ATTRIBUTE_VALUES, DS_O_PERSON),
OM_OID_DESC(DS_ATTRIBUTE_VALUES, DS_O_ORG_PERSON),
OM_NULL_DESCRIPTOR
};

/* Build up an array of object identifiers for the optional packages    */
/* to be negotiated.                                                  */

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

#endif /* THRADD_H */

```

Chapter 9. XDS/XOM Convenience Routines

This chapter describes functions which are available to XDS/XOM programmers to help simplify and speed up the development of XDS applications. The convenience functions target two main areas:

- Filling, comparing and extracting objects
- Converting objects to and from strings

Six convenience functions are provided:

- **dsX_extract_attr_values()**
- **omX_fill()**
- **omX_fill_oid()**
- **omX_extract()**
- **omX_string_to_object()**
- **omX_object_to_string()**

Refer to the *z/OS DCE Application Development Reference* for a detailed description of these functions.

To demonstrate the power of the convenience functions, the **teldir.c** example program from Chapter 7, “Example Application Programs” on page 159 is presented again in this chapter, but it has been modified to make use of these functions. The modified example program is called **teldir2.c**.

String Handling

The convenience functions provide the ability to specify OM objects in string format by means of abbreviations. These abbreviations are defined in the XOM object information file **xoischema**.

X.500 attribute types can be specified either as abbreviations or object identifier strings. The mapping of the attribute abbreviations and object identifier strings to BER encoded object identifiers and the associated attribute syntaxes is determined by the XOM object information module with the help of the **xoischema** file. For valid attribute abbreviations, please refer to the **xoischema** file in the following directory:

```
/opt/dce1ocal/etc
```

It is important that any schema changes to the DSA are reflected in the **xoischema** file.

The convenience functions are able to handle strings with special syntax. The strings can be broadly classified into the following:

- Strings representing GDS attribute information
- Strings representing structured GDS attribute information
- Strings representing a structured GDS attribute value
- Strings representing a Distinguished Name (DN)
- Strings representing expressions

Strings Representing GDS Attribute Information

Strings that represent GDS attribute information are used to associate the attributes with their values. They are of the form:

Attribute Type = Attribute Value

The attribute types can either be specified as abbreviations or object identifier strings. An object identifier string is defined as a series of digits separated by the . (dot) character. If attribute abbreviations are used, they are case insensitive. For example, specify **cn=schmid** or **85.4.3=schmid**.

In the case of attributes with **OM_S_OBJECT_IDENTIFIER_STRING** syntax, the attribute value can also be specified as an abbreviation string. For example, an object class for **Residential Person** can be specified as **OCL=REP** or **OCL='\x55\x06\x0A'**.

All leading and trailing whitespace (surrounding the attribute type, the = (equal sign) character, and the attribute value) is ignored.

The following are the reserved characters for such strings:

- ' Used to enclose the attribute values. If this character is used, all other reserved characters within the quoted string except the \ character are not interpreted. For example:
cn='henry mueller'
- ; Separates multiple values of a recurring attribute. All leading and trailing whitespace (surrounding the ; character) is ignored. For example:
TN=899898;979779
- = Associates the attribute with its value.
- \xnn Specifies hexadecimal data. After the \x, the next two characters are read as the hexadecimal value.
- \ Used to escape any of the other reserved characters.

Strings Representing Structured GDS Attribute Information

Strings that represent structured GDS attribute information are used to associate the structured attribute and its components with their values. They are of the form:

Structured Attribute Type = {Comp1 = Value, Comp2 = Value, ...}

The structured attribute type can either be specified as abbreviations or object identifier strings. An object identifier string is defined as a series of digits separated by the . (dot) character. If attribute abbreviations are used, they are case insensitive. *Comp1*, *Comp2*, and so on, are the components of the structured attribute. They should be specified as abbreviations. For example:

TXN={TN=977999, CC=345, AB=8444}

Recurring values for structured attributes can be specified with the help of the ; character. For example:

TXN={TN=977999, CC=345, AB=8444};{TN=123444,CC=345, AB=8444}

Recurring values for the components should be specified as:

TXN={TN=977999; 274424, CC=345, AB=8444}

If any of the components are further structured, they should be enclosed within braces as:

FTN={PA={FR=1,TD=1}, PN=67899}

All leading and trailing whitespace (surrounding the structured attribute type, the component abbreviation, the = (equal sign) character, the { (left brace) character, the , (comma) character and the } (right brace) character) is ignored.

Attributes and components with DN syntax should be specified as:

AON={/c=de/o=sni/ou=ap11/cn=mueller}

ACL={MPUB={INT=0, USR={/c=de/o=sni/cn=mueller, sn=schmid}}}

In the case of attributes with **OM_S_OBJECT_IDENTIFIER_STRING** syntax, the attribute value can also be specified as an abbreviation string. For example:

SG={OCL=REP} or SG={OCL='\x55\x06\x0A'}

The following are the reserved characters for strings with structured attribute information:

' Used to enclose the attribute values. If this character is used, all other reserved characters within the quoted string except the \ character are not interpreted. For example:

cn='henry mueller'

/ Specifies an attribute value with DN Syntax. For example:

AON = {/c=de/o=sni/ou=ap22/cn=mayer}

{ Indicates the start of a structured attribute value block.

} Indicates the end of a structured attribute value block.

, Separates the components of a structured attribute. For example:

TN=977999, CC=345, AB=8444

It can also be used to specify multiple AVAs in the case of attributes with DN syntax.

; Separates multiple values of a recurring attribute or the recurring components of the structured attribute. All leading and trailing whitespace (surrounding the attribute type, = character, { character, } character, the component abbreviation, the component value and the ; character) is ignored. For example:

TXN={TN=977999,CC=345,AB=8444};{TN=53533,CC=242,AB=44242}

= Associates the components with their value, and associates the components to the structured attribute.

\xnn Used to specify hexadecimal data. After the \x, the next two characters are read as the hexadecimal value.

\ Used to escape any of the other reserved characters.

Strings Representing a Structured GDS Attribute Value

Strings are used to represent the structured GDS attribute value. Only one structured attribute value can be specified. They are of the form:

Comp1 = Value, Comp2 = Value, ...

Comp1, Comp2, and so on are the components of the structured attribute. They should be specified as abbreviations. For example, to specify a value for **DS_C_TELEX_NBR** class, the string format is:

TN=977999, CC=345, AB=8444

Recurring values for the components can be specified as:

TN=977999; 274424, CC=345, AB=8444

If any of the components are further structured, they should be enclosed within braces and specified as:

FTP={FR=1,TD=1}, PN=67899

Components with DN syntax can be specified as:

MPUB={INT=0, USR={/c=de/o=sni/cn=mueller, sn=schmid}}

The reserved characters for such strings are the same as those for strings representing structured attribute information (as described in "Strings Representing Structured GDS Attribute Information" on page 210).

Strings Representing a Distinguished Name

Strings are used to represent the DN of the object. They are of the form:

/<Attribute Type> = <Naming Attribute Value> ...

or

/<Attribute Value>/<Attribute Value> ...

The attribute types can either be specified as abbreviations or object identifier strings. An object identifier string is defined as a series of digits separated by the . (dot) character. If attribute abbreviations are used, they are case insensitive. Multiple AVAs are represented by separating the naming attribute values with the , (comma) character.

The first RDN can also be specified as the DCE global root string */...*, which is a sequence of the / (slash) character followed by three dot characters. In this case, the */...* string is simply ignored and the rest of the string is processed. For example:

/c=de/o=sni/ou=ap11, l=munich/85.4.3=schmid

or

/c=us/o=osf/ou=abc/subsystems/server/xyz

or

`/.../c=us/o=osf/ou=abc/subsystems/server/xyz`

The first nonspace character should always be the / character. All leading and trailing whitespace (surrounding the / character, attribute type, the = character and the attribute value) is ignored.

Following are the reserved characters:

- ' Used to enclose the naming attribute values. If this character is used, all other reserved characters within the quoted string except the \ character are not interpreted. For example:
`cn='henry mueller'`
- / Used as a delimiter between RDNs.
- , Specifies multiple AVAs. All leading and trailing whitespace surrounding the , character is ignored. For example:
`/c=de/o=dbp/ou=dap11/cn=schmid, ou=ap11`
- = Associates the object with its naming attribute value.
- \xnn Used to specify hexadecimal data. After the \x, the next two characters are read as the hexadecimal value.
- \ Used to escape any of the other reserved characters.

Strings Representing Expressions

Strings are used to specify an SQL-like expression in a search operation. For example:

`(CN~=schmid) && (OCL=ORP || OCL=REP) && !(SN=ronnie)`

This is used to search for anybody who is an organizational person or a residential person, whose name approximately matches **schmid**, but whose surname is not **ronnie**.

Object identifiers can also be used instead of attribute abbreviations. The object identifier string is a series of numbers separated by the . (dot) character.

All leading and trailing whitespace (surrounding the attribute types, the operators, and the attribute values) is ignored.

If spaces are part of the attribute value, then the complete attribute value must be enclosed in quotes.

Additionally, the presence of an attribute can also be tested in either of the following ways:

`c = de && cn`
`c = de && cn = *`

The following are the reserved characters:

- ' Used to indicate the start and end of an attribute value string. It can be used when spaces are part of the data. If this character is used, all other reserved characters within the quoted string except the \ character are not interpreted. For example:
`OU=sni && cn='Henry Mueller' && tn=89989`
- / Used to specify an attribute value with DN Syntax. For example: **`AON = {/c=de/o=sni/ou=ap22/cn=mayer}`**
- = Used to associate the attribute with its value.
- && Used to logically AND two conditions.

- || Used to logically OR two conditions.
- ! Used to logically NEGATE a condition.
- ~= Used to specify phonetic matching during a search operation.
- > Used to match values greater than a specified value.
- >= Used to match values greater than or equal to a specified value.
- < Used to match values less than a specified value.
- <= Used to match values less than or equal to a specified value.
- Used to specify substrings during search.
- (Used for nesting of filters.
-) Used for nesting of filters.
- { Indicates the start of a structured attribute value block.
- } Indicates the end of a structured attribute value block.
- , Separates the components of a structured attribute. For example:
TN=977999, CC=345, AB=8444
 It can also be used to specify multiple AVAs in the case of attributes with DN syntax.
- \xnn Used to specify hexadecimal data. After the \x, the next two characters are read as the hexadecimal value.
- \ Used to escape any of the other reserved characters.

While evaluating complex expressions during search operations, the following precedence of operators prevail:

1. ()
2. !
3. &&
4. ||

The () operators have the highest precedence, and || the lowest.

Examples of strings handled by omX_string_to_object()

Following are examples of strings that can be handled by omX_string_to_object():

Example 1: To create a DS_C_DS_DN object (Root):

/ or /...

Example 2: To create other **DS_C_DS_DN** objects:

```
/c=de/o=sni/ou=ap11/cn=naik,sn=naik
/c=de/o=sni/ou=ap11/85.4.3=naik,sn=naik
/c=de/o=sni/ou=ap11/cn=naik,sn=na\x69k
/c=de/o=sni/ou=ap11/cn=naik,loc=Muenchen\,8000
/c=de/o=sni/ou=ap11/cn=naik,loc='Muenchen,8000'
/C = de / O = sni / Ou = ap11/CN=naik, SN=naik
```

Example 3: To create a **DS_C_DS_DN** object (DCE name):

```
/.../c=us/o=osf/ou=abc/subsystems/server/xyz
```

Example 4: To create a **DS_C_DS_RDN** object:

```
cn=naik,sn=naik
cn=naik,sn=na\x89k
CN = naik, SN = naik
```

Example 5: To create a **DS_C_DS_RDN** object (DCE name):

```
server
```

Example 6: To create a **DS_C_ATTRIBUTE** object (containing, for example, **Common-Name**):

```
cn=bhavesh naik
CN = bhavesh naik
85.4.3=bhavesh na\x89k
```

Example 7: To create a **DS_C_ATTRIBUTE** object (containing an object class with multiple values of **Residential-Person** and **Organizational-Person**):

```
OCL=REP;ORP
OCL = '\x55\x06\x0a' ; '\x55\x06\x07'
```

Example 8: To create a **DS_C_ATTRIBUTE** object (containing a GDS structured attribute like **Telex-Number** or **Owner**):

```
TXN={TN=12345,CC=678,AB=90}
TXN = { TN = 12345, CC = 678, AB = 90}
own={/c=de/o=sni/ou=ap11};{/c=de/o=sni/ou=ap22}
pa={pa='Wilhelm Riehl Str.85';'Munich')}
```

Example 9: To create a **DSX_C_GDS_ACL** object:

```
MPUB={INT=0, USR={/c=de/o=sni/cn=naik, sn=bhavesh}}
```

Example 10: To create a **DS_C_FILTER** object:

```
c
!c
C = de && CN = 'bha\x76esh naik'
c=de&&cn~=mueller
c = de && (cn = 'a*' || cn = b* || cn = c* )
ACL={MPUB={INT=0,USR={/c=de/o=sni/cn=naik, sn=bhavesh}}}
c = de || cn = *aa*bb*cc*
(cn ~=naik)&&((OCL=ORP)||(OCL=REP))&& !(SN='bhavesh naik')&&(L=*)
```

Example 11: Example of the error return when an erroneous string is supplied:

```
/c=de/o=sni,=de
```

The **OM_return_code** would be **OM_WRONG_VALUE_MAKEUP**.

The **error_type** would be **OMX_MISSING_ABBRV**.

The **error_position** would be 13.

Examples of strings returned by **omX_object_to_string()**

Following are examples of strings returned by **omX_object_to_string()**:

Example 1: If a **DS_C_DS_DN** object is supplied:

```
/
/C=de/O=sni/OU=ap11/CN=naik,SN=naik
/C=de/O=sni/OU=ap11/CN=naik,LOC=Muenchen\,8000
```

Example 2: If a **DS_C_DS_RDN** object is supplied:

```
CN=naik,SN=naik
server
```

Example 3: If a **DS_C_ATTRIBUTE** object is supplied:

```
CN=bhavesh naik
OCL=REP;ORP
TXN={AB=90,CC=678,TN=12345}
OWN={/C=de/O=sni/OU=ap11};{/C=de/O=sni/OU=ap22}
```

Example 4: If a **DSX_C_GDS_ACL** object is supplied:

```
MPUB={INT=0,USR={/C=de/O=sni/CN=naik,SN=bhavesh}}
```

Example 5: If a **DS_C_NAME_ERROR** object is supplied with **DS_PROBLEM** of **DS_E_NO_SUCH_OBJECT**:

The specified name does not match the name of any object in the directory

Example 6: If a **DS_C_ATTRIBUTE_ERROR** object is supplied with **DS_C_ATTRIBUTE_PROBLEM** containing **DS_E_ATTRIBUTE_OR_VALUE_EXISTS**:

An attempt is made to add an attribute or value that already exists.

The teldir2.c Program

The **teldir2.c** file is a program that performs the same functionality as **teldir.c**. Please refer to Chapter 7 for a complete description of the program's functionality, including outputs. The purpose of **teldir2.c** is to show how the XDS/XOM convenience functions can be used to reduce the complexity of a real application.

The source code for this program can be found in directory `/usr/lpp/dce/examples/xdsxom/teldir2.c`.

The program consists of the following steps:

0. Initialize public OM objects.
1. Examine the command-line argument to determine the type of operation (read, add, or delete entry) that the user wants to perform.
2. Initialize a workspace.
3. Pull in the packages with the required XDS features.
4. Prompt the user for the name entry on which the operation will be performed.
5. Convert the DCE-formatted user input string to an XDS object name.
6. Bind (without credentials) to the default server.
7. Perform the requested operation (read, add, or delete entry).
9. Unbind from the server.
10. Shutdown the workspace, releasing resources back to the system.

The **teldir2.c** program differs from the **teldir.c** program primarily because it makes use of the XOM convenience routines. It shows how these routines can simplify a program that needs to use the XDS/XOM APIs by aiding both the generation and parsing of OM objects. The following list of points summarizes the program differences resulting from this approach:

- **Step 0** replaces the declaration of static OM objects **xdsSurname**, **xdsPhoneNum**, and **xdsAttributesToAdd** performed in **teldir.c** with dynamically created OM objects that are filled in at the beginning of the program's processing. The **omX_fill()** and **omX_fill_oid()** convenience routines are used to initialize these OM objects.
- The **extractValue()** function defined in **teldir2.c** is quite different than the **extractValue()** function defined in **teldir.c**. In **teldir2.c**, XOM convenience routine **dsX_extract_attr_values()** is used to aid in parsing the **DS_C_READ_RESULT** OM object passed into **extractValue()**. The first **om_get()** call in **extractValue()** extracts from the **DS_C_READ_RESULT** OM object a pointer to the **DS_C_ENTRY_INFO** OM object. This pointer is then passed to the **dsX_extract_attr_values()** function. In addition, the **dsX_extract_attr_values()** function takes an object identifier as input. This object identifier indicates which attribute in the **DS_C_ENTRY_INFO** OM object should be extracted.

The use of the **dsX_extract_attr_values()** XOM convenience routine replaces three **om_get()** function calls in the **teldir.c** implementation.

- The **stringToXdsName()** function defined in **teldir2.c** is also very different (and much simpler) than in that in **teldir.c**. In **teldir2.c** the **omX_string_to_object()** XOM convenience routine is used to replace all the manual parsing done by the **stringToXdsName()** function defined in **teldir.c**. Furthermore, two routines that support this parsing in **teldir.c**, **numNamePieces()** and **splitNamePiece()** are not needed and not included in **teldir2.c**.
- The **handleDSError()** function defined in **teldir2.c** makes use of the **omX_object_to_string()** XOM convenience routine to obtain a textual error message based on a **DS_C_ERROR** OM object passed in to the **omX_object_to_string()** XOM convenience routine.
- When the add option is specified in **teldir2.c**, the **omX_string_to_object()** XOM convenience routine is used to create a **DS_C_ATTRIBUTE** OM object. This OM object is then pointed to from the **xdsAttributesToAdd** OM object (which is of OM class **DS_C_ATTRIBUTE_LIST**). The **omX_fill()** XOM convenience routine is used to modify the **xdsAttributesToAdd** public OM object.

The teldir2.c Code

The following code is a listing of the **teldir2.c** program:

```
/*
 * This sample program behaves like a simple telephone directory.
 * It permits a user to add, read or delete entries in a GDS
 * namespace or to a CDS namespace in any local or remote DCE cell
 * (assuming that permissions are granted by the ACLs).
 *
 * Each entry is of class Organizational-Person and simply contains
 * a person's surname and their phone number.
 *
 * The addition of an entry is followed by a read to verify that the
 * information was entered properly.
 *
 * All valid names should begin with one of the following symbols:
 *   /...      Fully qualified name (from global root).
 *             e.g. /.../C=de/O=sni/OU=ap/CN=klaus
 *
 *   /.:      Partially qualified name (from local cell root).
 *             e.g. /.:/brad/sni/com
 *
 * This program demonstrates the following techniques:
 * - Using omX_fill and omX_fill_oid to update non-static XDS
 *   public objects. See xdsObjectClass and xdsAttributesToAdd.
 * - Using omX_string_to_object to create DS_C_ATTRIBUTE type XDS
 *   objects from their string representations. See xdsSurname
 *   and xdsPhoneNum.
 * - Using omX_string_to_object to create a DS_C_DS_DN type XDS
 *   object which represents an object name. See the function
 *   stringToXdsName() below.
 * - Getting the value of an attribute from an object read from the
 *   namespace using ds_read(). See the function extractValue() below.
 *   omX_extract and dsX_extract_attr_values are used.
 * - Getting a string representation of the numeric DS_PROBLEM error
 *   value which is returned within a DS_C_ERROR private object when an
 *   error occurs during one of the XDS calls. See the function
 *   handleDSError() below.
 */
```

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <xom.h>
#include <xomext.h>
#include <xds.h>
#include <xdsext.h>
#include <xdsbdc.h>
#include <xdsfds.h>
#include <xdsfds.h>

OM_EXPORT( DS_BASIC_DIR_CONTENTS_PKG )
OM_EXPORT( DS_A_COMMON_NAME )
OM_EXPORT( DS_A_COUNTRY_NAME )
OM_EXPORT( DS_A_LOCALITY_NAME )
OM_EXPORT( DS_A_OBJECT_CLASS )
OM_EXPORT( DS_A_ORG_UNIT_NAME )
OM_EXPORT( DS_A_ORG_NAME )
OM_EXPORT( DS_A_SURNAME )
OM_EXPORT( DS_A_PHONE_NBR )
OM_EXPORT( DS_A_TITLE )
OM_EXPORT( DS_C_ATTRIBUTE )
OM_EXPORT( DS_C_ATTRIBUTE_LIST )
OM_EXPORT( DS_A_STATE_OR_PROV_NAME )
OM_EXPORT( DS_C_AVA )
OM_EXPORT( DS_C_DS_DN )
OM_EXPORT( DS_C_DS_RDN )
OM_EXPORT( DS_C_ENTRY_INFO_SELECTION )
OM_EXPORT( DS_O_ORG_PERSON )
OM_EXPORT( DS_O_PERSON )
OM_EXPORT( DS_O_TOP )
OM_EXPORT( DSX_TYPELESS_RDN ) /* For "typeless" pieces of a name, as */
                               /* found in cells with bind-style names */
                               /* and in the CDS namespace. */

#define MAX_NAME_LEN 1024

/* These values can be found in the Chapter
   "Directory Class Definitions". */
/* (One byte must be added for the null terminator.) */
#define MAX_PHONE_LEN 33
#define MAX_SURNAME_LEN 66

/*****
 * Static XDS objects.
 *****/
/*
 * To hold the list of attributes we want to read.
 */
static OM_descriptor xdsAttributeSelection[] = {

    /* This is an entry information selection. */
    OM_OID_DESC( OM_CLASS, DS_C_ENTRY_INFO_SELECTION ),

    /* Get all attributes. */
    { DS_ALL_ATTRIBUTES, OM_S_BOOLEAN, OM_FALSE },

```

```

/* These are the ones we want to read. */
OM_OID_DESC( DS_ATTRIBUTES_SELECTED, DS_A_SURNAME ),
OM_OID_DESC( DS_ATTRIBUTES_SELECTED, DS_A_PHONE_NBR ),

/* Give us both the types and their values. */
{ DS_INFO_TYPE, OM_S_ENUMERATION, DS_TYPES_AND_VALUES },

/* Null terminator */
OM_NULL_DESCRIPTOR
};

/*****
 * showUsage()
 *   Display "usage" information.
 *****/
void
showUsage(
    char *    cmd          /* In--Name of command being called */
)
{
    fprintf( stderr, "\nusage:  %s [option]\n\n", cmd );
    fprintf( stderr, "option:  a : add an entry\n" );
    fprintf( stderr, "        r : read an entry\n" );
    fprintf( stderr, "        d : delete an entry\n" );
} /* end showUsage() */

/*****
 * extractValue()
 *   Pulls the value of a particular attribute from a private object
 *   that was received using ds_read().
 *   Returns:
 *       OM_SUCCESS      If successful.
 *       OM_NO_SUCH_OBJECT  If no values for the attribute
 *                           were found.
 *       other           Any value returned by one of the
 *                           omX_extract() and dsX_extract_attr_values()
 *                           calls.
 *****/
OM_return_code
extractValue(
    OM_private_object  object,          /* In--Object to extract from */
    OM_object_identifier attribute,     /* In--Attribute to extract */
    char *            value            /* Out--Value found */
)
{
    int                i;
    OM_return_code     omStatus;
    OM_descriptor      *attrValue;
    OM_public_object   outputValues;
    OM_value_position  totalValues;
    OM_type            navigationPath[1];
    OM_type            typeList[2];
    OM_private_object  entryInfo;

```

```

/*
 * the DS_ENTRY object is pointed to by the "root"
 * DS_C_READ_RESULT object which is returned by ds_read().
 */
navigationPath[0] = 0;

/*
 * extract only the DS_ENTRY OM object which points
 * to a DS_C_ENTRY_INFO object.
 */
typeList[0] = DS_ENTRY;
typeList[1] = 0;

/*
 * Get the DS_C_ENTRY_INFO entry from the object returned
 * by ds_read().
 */
omStatus = omX_extract( object,
                       navigationPath,
                       OM_EXCLUDE_ALL_BUT_THESE_TYPES
                       | OM_EXCLUDE_SUBOBJECTS,
                       typeList,
                       OM_TRUE, /* local_strings == OM_TRUE */
                       0,
                       0,
                       &entryInfo,
                       &totalValues);
if( omStatus != OM_SUCCESS ) {
    fprintf( stderr, "omX_extract( entry ) returned error %d\n",
            omStatus );
    return( omStatus );
}

/*
 * dsX_extract_attr_values() takes either a DS_C_ENTRY_INFO
 * or DS_C_ATTRIBUTE_LIST object as input and will return
 * the attribute values associated with the input attribute type.
 */
omStatus = dsX_extract_attr_values( (OM_private_object)
                                   entryInfo->value.object.object,
                                   attribute, /* attribute type OID */
                                   OM_TRUE, /* local_strings == OM_TRUE */
                                   &outputValues,
                                   &totalValues);
if( omStatus != OM_SUCCESS ) {
    fprintf( stderr, "dsX_extract_attr_values( entry ) returned error %d\n",
            omStatus );
    return( omStatus );
}
if( totalValues <= 0 ) { /* Make sure something was returned */
    fprintf( stderr,
            "Number of descriptors returned by dsX_extract_attr_values( entry ) was %d\n",
            totalValues );
    return( OM_NO_SUCH_OBJECT );
}

/*
 * Copy the value(s) into the buffer for return to the caller.

```

```

    */
    attrValue = outputValues;
    for(i=0; i<totalValues; i++) {
        if ( i != 0 ) {
            *value++ = ';'; *value++ = ' ';
        }
        strncpy( value, attrValue->value.string.elements,
            attrValue->value.string.length );
        value += attrValue->value.string.length;
        attrValue++;
    }
    *value = '\0';

    /*
    * Free up the resources we don't need any more and return.
    */
    om_delete( entryInfo );
    om_delete( outputValues );
    return( OM_SUCCESS );
} /* end extractValue() */

/*****
* stringToXdsName()
* Converts a string that is a DCE name to an XDS name object (class
* DS_C_DS_DN). Returns one of the following:
* OM_SUCCESS If successful.
* OM_WRONG_VALUE_MAKEUP If the omX_string_to_object() fails.
*
* The name will be allocated by omX_string_to_object() as a private
* OM object in the workspace supplied. As such, it will be cleaned
* up when a ds_shutdown() is called.
*****/
OM_return_code
stringToXdsName(
    char * origString, /* In--String name to be converted */
    OM_workspace * workspace, /* In--Workspace to use */
    OM_object * xdsNameObj /* Out--Pointer to XDS name object */
)
{
    char nameString[MAX_NAME_LEN]; /* slightly modified
                                     origString */
    char *cellName; /* output of dce_cf_get_cell_name() */
    unsigned long rc; /* output of dce_cf_get_cell_name() */
    OM_string inputString; /* input to omX_string_to_object() */
    OM_return_code omStatus; /* return from omX_string_to_object() */
    OM_integer errorPosition; /* output of omX_string_to_object() */
    OM_integer errorType; /* output of omX_string_to_object() */

    omStatus = OM_SUCCESS;

    if ( strcmp("././",origString,4) == 0 ) {
        dce_cf_get_cell_name( &cellName, &rc );
        if ( rc == 0 ) {
            strcpy(nameString, cellName);
            strcat(nameString, "/");
            strcat(nameString, origString+4);

```

```

        } else {
            omStatus = OM_SYSTEM_ERROR;
        }
    } else {
        strcpy(nameString, origString);
    }

    if ( omStatus == OM_SUCCESS ) {
        inputString.length = strlen(nameString);
        inputString.elements = nameString;

        omStatus = omX_string_to_object( workspace,
                                        &inputString,
                                        DS_C_DS_DN,
                                        OM_TRUE, /* local_strings == OM_TRUE */
                                        xdsNameObj, /* output - generated name object */
                                        &errorPosition,
                                        &errorType);

        if ( omStatus != OM_SUCCESS ) {
            fprintf(stderr,
                "Error parsing supplied name, error_position:%d, error_type:%d\n",
                errorPosition, errorType);
        }
    } else {
        fprintf(stderr, "Error parsing supplied name\n");
    }

    return(omStatus);
} /* end stringToXdsName() */

```

```

/*****
 * handleDSError()
 *   Extracts the error number from a DS_status return code, prints it
 *   in an error message, then terminates the program.
 *****/
void
handleDSError(
    char *      header, /* In--Name of function whose return code */
                /* is being checked */
    DS_status   returnCode /* In--Return code to be checked */
)
{
    OM_return_code   omStatus;
    OM_string        outputString;

    if ( returnCode == DS_SUCCESS ) {
        /* do nothing */
    } else if ( (returnCode == DS_NO_WORKSPACE) ||
                (returnCode == DS_NO_WORKSPACE) ) {
        printf( "%s failed due to a malformed workspace\n", header);
    } else {

        outputString.length = MAX_NAME_LEN;
        outputString.elements = malloc(MAX_NAME_LEN);

        if ( outputString.elements != NULL ) {

```

```

/*
 * A DS_status return code is an object. It will be one of the
 * subclasses of the class DS_C_ERROR. What we want from it is
 * the value of the attribute DS_PROBLEM. The omX_object_to_string()
 * routine will return a character string containing a text message
 * describing the error.
 */
omStatus = omX_object_to_string( returnCode,
                                OM_TRUE, /* local_strings == OM_TRUE */
                                &outputString);

/*
 * Make sure we successfully extracted the message text.
 */
if( omStatus == OM_SUCCESS ) {
    printf( "%s returned error %s\n", header,
            outputString.length, outputString.elements);
} else {
    printf( "%s failed for unknown reason\n", header );
}

    free( outputString.elements );
} else {
    printf( "handleDSError() could not obtain storage\n");
}
}

exit( 1 );
}

/*****
 * Main program
 */
void
main(
    int     argc,
    char *  argv[]
)
{
    DS_status      dsStatus;
    OM_sint        invokeID;
    char           newName[MAX_NAME_LEN];
    char           newPhoneNum[MAX_PHONE_LEN];
    char           newSurname[MAX_SURNAME_LEN];
    char           objectStringElements[MAX_NAME_LEN];
    OM_string      objectString;
    OM_return_code omStatus;
    char           phoneNumRead[MAX_PHONE_LEN];
    int            rc = 0;
    OM_private_object readResult;
    OM_private_object session;
    char           surnameRead[MAX_SURNAME_LEN];
    OM_object      xdsName;
    OM_workspace   xdsWorkspace;
    char           operation;
    OM_integer     errorPosition;

```

```

OM_integer      errorType;
DS_feature     featureList[2];

/*
 * STEP 0
 *
 * Initialize the OM public objects to be used.
 */

OM_descriptor xdsObjectClass[4];
OM_private_object xdsSurname;
OM_private_object xdsPhoneNum;
OM_descriptor xdsAttributesToAdd[5];

/*
 * Set up the DS_feature list. This is used to identify which
 * packages we need for this program. We only need
 * the basic package because we are not doing anything fancy with
 * session parameters, etc.
 */
featureList[0].feature = DS_BASIC_DIR_CONTENTS_PKG;
featureList[0].activated = OM_TRUE;
featureList[1].feature.length = 0;
featureList[1].feature.elements = NULL;

/* set up a OM_string structure with storage to hold data */
objectString.length = MAX_NAME_LEN;
objectString.elements = objectStringElements;

/* Fill in the parts of the xdsObjectClass OM object */
omX_fill_oid( OM_CLASS, DS_C_ATTRIBUTE, &xdsObjectClass[0]);
omX_fill_oid( DS_ATTRIBUTE_TYPE, DS_A_OBJECT_CLASS, &xdsObjectClass[1]);
omX_fill_oid( DS_ATTRIBUTE_VALUES, DS_O_ORG_PERSON, &xdsObjectClass[2]);
omX_fill( OM_NO_MORE_TYPES, OM_S_NO_MORE_SYNTAXES, 0, OM_ELEMENTS_UNSPECIFIED,
          &xdsObjectClass[3]);

/* Fill in the non-variant parts of the xdsAttributesToAdd OM object */
omX_fill_oid( OM_CLASS, DS_C_ATTRIBUTE_LIST, &xdsAttributesToAdd[0]);
omX_fill( DS_ATTRIBUTES, OM_S_OBJECT, 0, xdsObjectClass,
          &xdsAttributesToAdd[1]);
omX_fill( OM_NO_MORE_TYPES, OM_S_NO_MORE_SYNTAXES, 0, OM_ELEMENTS_UNSPECIFIED,
          &xdsAttributesToAdd[4]);

/* STEP 1
 *
 * Examine command-line argument.
 */
operation = *argv[1];
if ( (operation != 'r') && (operation != 'a') && (operation != 'd') ) {
    showUsage( argv[0] );
    exit( 1 );
}

/* STEP 2
 *

```

```

    * Initialize the XDS workspace.
    */
xdsWorkspace = ds_initialize( );
if( xdsWorkspace == NULL ) {
    fprintf( stderr, "ds_initialize() failed\n" );
    exit( 1 );
}

/* STEP 3
 *
 * Pull in the packages that contain the XDS features we need.
 */
dsStatus = ds_version( featureList, xdsWorkspace );
if( dsStatus != DS_SUCCESS )
    handleDSError( "ds_version()", dsStatus );

/* STEP 4
 *
 * Find out what name the user wants to use in the namespace and
 * convert it to an XDS object. We do this conversion dynamically
 * (not using static structures defined at the top of the program)
 * because we don't know how long the name will be.
 */
switch( operation ) {
case 'r' :
    printf( "What name do you want to read? " );
    break;
case 'a' :
    printf( "What name do you want to add? " );
    break;
case 'd' :
    printf( "What name do you want to delete? " );
    break;
}

/* STEP 5 */

gets( newName );
omStatus = stringToXdsName( newName, xdsWorkspace, &xdsName );
if( omStatus != OM_SUCCESS ) {
    fprintf( stderr,
            "stringToXdsName() failed with OM error %d\n", omStatus );
    exit( 1 );
}

if ( operation == 'a' ) { /* add operation requires additional input */
    /*
     * Get the person's real name from the user and, using
     * omX_string_to_object(), create the xdsSurname OM object which
     * will contain the data.
     *
     * We are requiring a name, so we will loop until we get one.
     */
    do {
        printf( "What is this person's surname? " );
        gets( newSurname );
    } while ( *newSurname == '\0' );
}

```

```

sprintf( (char *) (objectString.elements), "SN=%s", newSurname);
objectString.length=strlen(objectString.elements);

omStatus = omX_string_to_object( xdsWorkspace,
                                &objectString,
                                DS_C_ATTRIBUTE,
                                OM_TRUE, /* local_strings == OM_TRUE */
                                &xdsSurname,
                                &errorPosition,
                                &errorType);

omX_fill( DS_ATTRIBUTES, OM_S_OBJECT, 0, xdsSurname,
          &xdsAttributesToAdd[2]);

/*
 * Get the person's phone number from the user and, using
 * omX_string_to_object(), create the xdsPhoneNum OM object which
 * will contain the data.
 *
 * A phone number is not required, so if none is given we will create
 * the xdsPhoneNum OM object with "unlisted" as the telephone number.
 */
printf( "What is this person's phone number? ");
gets( newPhoneNum );
if( *newPhoneNum != '\0' ) {
    sprintf( (char *) (objectString.elements), "TN=%s",
            newPhoneNum);
} else {
    strcpy( (char *) (objectString.elements), "TN=unlisted");
}
objectString.length=strlen(objectString.elements);

omStatus = omX_string_to_object( xdsWorkspace,
                                &objectString,
                                DS_C_ATTRIBUTE,
                                OM_TRUE, /* local_strings == OM_TRUE */
                                &xdsPhoneNum,
                                &errorPosition,
                                &errorType);

omX_fill( DS_ATTRIBUTES, OM_S_OBJECT, 0, xdsPhoneNum,
          &xdsAttributesToAdd[3]);
}

/* STEP 6
 *
 * Open the session with the namespace:
 * bind (without credentials) to the default server.
 */
dsStatus = ds_bind( DS_DEFAULT_SESSION, xdsWorkspace, &session );
if( dsStatus != DS_SUCCESS )
    handleDSError( "ds_bind()", dsStatus );

/* STEP 7 */

switch( operation ) { /* perform the requested operation */

```

```

/*
 * Add entry to the namespace. The xdsSurname and xdsPhoneNum
 * objects are already contained within an attribute list object
 * (xdsAttributesToAdd).
 */
case 'a' :
    dsStatus = ds_add_entry( session, DS_DEFAULT_CONTEXT, xdsName,
                            xdsAttributesToAdd, &invokeID );
    if( dsStatus != DS_SUCCESS )
        handleDSError( "ds_add_entry()", dsStatus );

    break;
/* FALL THROUGH */

/*
 * Read the entry of the name supplied.
 */
case 'r' :
    dsStatus = ds_read( session, DS_DEFAULT_CONTEXT, xdsName,
                       xdsAttributeSelection, &readResult, &invokeID );
    if( dsStatus != DS_SUCCESS )
        handleDSError( "ds_read()", dsStatus );

    /*
     * Get each attribute from the object read and print them.
     */
    omStatus = extractValue( readResult, DS_A_SURNAME, surnameRead );
    if( omStatus != OM_SUCCESS ) {
        printf( "** Surname could not be read\n" );
        strcpy( surnameRead, "(unknown)" );
        rc = 1;
    }
    omStatus = extractValue( readResult, DS_A_PHONE_NBR, phoneNumRead );
    if( omStatus != OM_SUCCESS ) {
        printf( "** Phone number could not be read\n" );
        strcpy( phoneNumRead, "(unknown)" );
        rc = 1;
    }
    printf( "The phone number for %s is %s.\n",
           surnameRead, phoneNumRead );

    break;

/*
 * delete the entry from the namespace.
 */
case 'd' :
    dsStatus = ds_remove_entry( session, DS_DEFAULT_CONTEXT, xdsName,
                                &invokeID );
    if( dsStatus != DS_SUCCESS )
        handleDSError( "ds_remove_entry()", dsStatus );
    else
        printf( "The entry has been deleted.\n" );
    break;
}

```

```
/*
 * Clean up and exit.
 */
/* STEP 8 */
dsStatus = ds_unbind( session );
if( dsStatus != DS_SUCCESS )
    handleDSError( "ds_unbind()", dsStatus );

/* STEP 9 */
dsStatus = ds_shutdown( xdsWorkspace );
if( dsStatus != DS_SUCCESS )
    handleDSError( "ds_shutdown()", dsStatus );

exit( rc );
} /* end main() */
```


Part 4. XDS/XOM Supplementary Information

This part of the book consists mostly of tables of values for the data structures used by the XDS and XOM application programming interfaces, which are the interfaces used to directly access the DCE Directory Service. These chapters supplement the information on the XDS and XOM function calls which are described in the *z/OS DCE Application Development Reference*.

Chapter 10. XDS Interface Description	233	DS_C_EXT	258
XDS Conformance to Standards	233	DS_C_FILTER	259
The XDS Functions	234	DS_C_FILTER_ITEM	260
The XDS Negotiation Sequence	235	DS_C_LIBRARY_ERROR	261
The session Parameter	235	DS_C_LIST_INFO	262
The context Parameter	236	DS_C_LIST_INFO_ITEM	263
The XDS Function Arguments	236	DS_C_LIST_RESULT	264
Attribute and Attribute Value Assertion	237	DS_C_NAME	264
The Entry-Information-Selection Parameter	237	DS_C_NAME_ERROR	265
The name Parameter	238	DS_C_OPERATION_PROGRESS	265
XDS Function Call Results	238	DS_C_PARTIAL_OUTCOME_QUAL	266
The invoke-id Parameter	238	DS_C_PRESENTATION_ADDRESS	267
The result Parameter	238	DS_C_READ_RESULT	268
The DS_status Return Value	239	DS_C_REFERRAL	268
Synchronous Operations	239	DS_C_RELATIVE_NAME	268
Security and XDS	240	DS_C_SEARCH_INFO	268
Other Features of the XDS Interface	240	DS_C_SEARCH_RESULT	269
Automatic Connection Management	240	DS_C_SECURITY_ERROR	270
Automatic Continuation and Referral Handling	240	DS_C_SERVICE_ERROR	270
		DS_C_SESSION	271
		DS_C_SYSTEM_ERROR	272
		DS_C_UPDATE_ERROR	273
Chapter 11. XDS Class Definitions	241		
Introduction to OM Classes	241	Chapter 12. Basic Directory Contents	
XDS Errors	241	Package	275
OM Class Hierarchy	242	Selected Attribute Types	275
DS_C_ABANDON_FAILED	244	Selected Object Classes	282
DS_C_ACCESS_POINT	245	OM Class Hierarchy	283
DS_C_ADDRESS	245	DS_C_FACSIMILE_TELEPHONE_NUMBER	284
DS_C_ATTRIBUTE	245	DS_C_POSTAL_ADDRESS	284
DS_C_ATTRIBUTE_ERROR	246	DS_C_SEARCH_CRITERION	285
DS_C_ATTRIBUTE_LIST	246	DS_C_SEARCH_GUIDE	286
DS_C_ATTRIBUTE_PROBLEM	247	DS_C_TELETEX_TERMINAL_IDENTIFIER	286
DS_C_AVA	248	DS_C_TELEX_NUMBER	287
DS_C_COMMON_RESULTS	248		
DS_C_COMMUNICATIONS_ERROR	248	Chapter 13. Strong Authentication	
DS_C_COMPARE_RESULT	249	Package	289
DS_C_CONTEXT	249	SAP Attribute Types	289
DS_C_CONTINUATION_REF	252	Strong Authentication Package Object	
DS_C_DS_DN	253	Classes	291
DS_C_DS_RDN	253	OM Class Hierarchy	291
DS_C_ENTRY_INFO	254	DS_C_ALGORITHM_IDENT	291
DS_C_ENTRY_INFO_SELECTION	254	DS_C_CERT	292
DS_C_ENTRY_MOD	255	DS_C_CERT_LIST	293
DS_C_ENTRY_MOD_LIST	256	DS_C_CERT_PAIR	293
DS_C_ERROR	256		

DS_C_CERT_SUBLIST	294	Relationship to ASN.1 Type Constructors . . .	328
DS_C_SIGNATURE	294		
Chapter 14. MHS Directory User Package	297	Chapter 18. XOM Service Interface	331
MDUP Attribute Types	297	Standards Conformance	331
MDUP Object Classes	299	XOM Data Types	331
MDUP OM Class Hierarchy	300	OM_boolean	333
MH_C_OR_ADDRESS	300	OM_descriptor	333
MH_C_OR_NAME	310	OM_enumeration	334
DS_C_DL_SUBMIT_PERMS	310	OM_exclusions	334
		OM_integer	335
		OM_modification	335
		OM_object	335
Chapter 15. Global Directory Service		OM_object_identifier	335
Package	313	OM_private_object	337
GDSP Attribute Types	313	OM_public_object	337
GDSP Object Classes	316	OM_return_code	337
GDSP OM Class Hierarchy	316	OM_string	338
DSX_C_GDS_ACL	316	OM_syntax	339
DSX_C_GDS_ACL_ITEM	317	OM_type	339
DSX_C_GDS_CONTEXT	318	OM_type_list	340
DSX_C_GDS_SESSION	321	OM_value	340
		OM_value_length	341
		OM_value_position	341
		OM_workspace	341
Chapter 16. Distributed Management		XOM Functions	341
Environment Support	323	XOM Return Codes	343
DME Attribute Types	323		
DME Object Classes	324		
		Chapter 19. Object Management Package	347
Chapter 17. Information Syntaxes	325	Class Hierarchy	347
Syntax Templates	325	Class Definitions	347
Syntaxes	325	OM_C_ENCODING	347
Strings	326	OM_C_EXTERNAL	348
Representation of String Values	327	OM_C_OBJECT	349
Relationship to ASN.1 Simple Types	327		
Relationship to ASN.1 Useful Types	327		
Relationship to ASN.1 Character String			
Types	328		

Chapter 10. XDS Interface Description

The XDS interface consists of a number of functions, together with many OM classes of OM objects, which are used as the arguments and results of the functions. Both the functions and the OM objects are based closely on the Abstract Service that is specified in the standards. (See *The Directory: Abstract Service Definition*, ISO 9594-3, CCITT X.511.)

The interface models the directory interactions as service requests made through a number of interface functions, which take a number of input *parameters*. Each valid request causes an **operation** within the Directory Service, which eventually returns a *status* and any result of the operation.

All interactions between the user and the Directory Service belong to a **session**, which is represented by an OM object passed as the first parameter to most interface functions.

The other parameters to the functions include a **context** and various service-specific parameters. The context includes a number of parameters that are common to many functions, and that seldom change from operation to operation.

Each of the components of this model are described in this chapter along with other features of the interface, such as security.

XDS Conformance to Standards

The XDS interface defines an API that application programs can use to access the underlying directory service. The DCE XDS API conforms to the *X/Open CAE Specification, API to Directory Services (XDS)*, (November 1991).

The DCE XDS implementation supports the following features:

- A synchronous interface. Asynchronous operations are not supported.
- All synchronous interface functions are supported. The two asynchronous-specific functions are handled as follows:
 - ds_abandon()** This call does not issue a Directory Service abandon operation. It returns with a **DS_C_ABANDON_FAILED (DS_E_TOO_LATE)** error.
 - ds_receive_result()** This call returns **DS_SUCCESS** with the *completion_flag_return* parameter set to **DS_NO_OUTSTANDING_OPERATION**.
- The **ds_search()** and **ds_modify_rdn()** routines are not supported by z/OS DCE. These routines return **DS_E_UNAVAILABLE** if a GDS name is used in the call. A **DS_E_UNWILLING_TO_PERFORM** is returned if a CDS name is used.
- Automatic connection management is not provided. The **ds_bind()** and **ds_unbind()** functions always try to set up and release Directory Service connections immediately.
- The **DS_FILE_DESCRIPTOR** attribute of the **DS_C_SESSION** object is not used.
- The default values for OM attributes in the **DS_C_CONTEXT** and **DS_C_SESSION** object are described in Chapter 11, “XDS Class Definitions” on page 241.

DCE XDS supports four packages: one is mandatory, and three are optional. Use of the optional packages is negotiated using **ds_version()**. The packages are as follows:

- The Directory Service Package (as defined in Chapter 11, “XDS Class Definitions” on page 241), which also includes the errors (as defined in “XDS Errors” on page 241). This package is mandatory.

- The Basic Directory Contents Package (as defined in Chapter 12, “Basic Directory Contents Package” on page 275). This package is optional.
- The Global Directory Service Package. This package is optional.
- The MHS Directory User Package. This package is optional.

Note: You can use any of the above packages, however, the objects and attributes supported by the Global Directory Service Package and the MHS Directory User package are complex and suited for GDS and mail objects in GDS. z/OS DCE does not support GDS.

None of the OM classes defined in the above packages are encodable. Thus, DCE XDS application programmers do not require use of the XOM functions **om_encode()** and **om_decode()**, which are not supported by the DCE XOM API.

The XDS Functions

As mentioned already, the standards define Abstract Services that requestors use to interact with the directory. Each of these Abstract Services maps to a single function call, and the detailed specifications are given in the *z/OS DCE Application Development Reference*. The services and the function calls to which they map are as follows:

- **DirectoryBind** (maps to **ds_bind()**)
- **DirectoryUnbind** (maps to **ds_unbind()**)
- **Read** (maps to **ds_read()**)
- **Compare** (maps to **ds_compare()**)
- **Abandon** (maps to **ds_abandon()**)
- **List** (maps to **ds_list()**)
- **Search** (maps to **ds_search()**)
- **AddEntry** (maps to **ds_add_entry()**)
- **RemoveEntry** (maps to **ds_remove_entry()**)
- **ModifyEntry** (maps to **ds_modify_entry()**)
- **ModifyRDN** (maps to **ds_modify_rdn()**)

There is a function called **ds_receive_result()**, which has no counterpart in the Abstract Service. It is used with asynchronous operations. (See the *z/OS DCE Application Development Reference* for information on how the asynchronous functions **ds_abandon()** and **ds_receive_result()** are handled by the DCE XDS API.)

The **ds_initialize()**, **ds_shutdown()**, and **ds_version()** functions are used to control the XDS API and do not initiate any directory operations.

The interface functions are summarized in Table 20.

Table 20 (Page 1 of 2). The XDS Interface Functions

Name	Description
ds_abandon()	Abandons the result of a pending, asynchronous operation. This function is not supported. (See the <i>z/OS DCE Application Development Reference</i> .)
ds_add_entry()	Adds a leaf entry to the DIT.

Table 20 (Page 2 of 2). The XDS Interface Functions

Name	Description
ds_bind()	Opens a session with a DUA that, in turn, connects to a DSA.
ds_compare()	Compares a purported attribute value with the attribute value stored in the DIB for a particular entry.
ds_initialize()	Initializes the XDS interface.
ds_list()	Enumerates the names of the immediate subordinates of a particular directory entry.
ds_modify_entry()	Automatically modifies a directory entry.
ds_modify_rdn()	Changes the RDN of a leaf entry. Not supported on z/OS DCE.
ds_read()	Queries information on a particular directory entry by name.
ds_receive_result()	Retrieves the result of an asynchronously executed function. This function is not supported. (See <i>z/OS DCE Application Development Reference</i> .)
ds_remove_entry()	Removes a leaf entry from the DIT.
ds_search()	Finds entries of interest in a portion of the directory information tree. Not supported on z/OS DCE.
ds_shutdown()	Discards a workspace.
ds_unbind()	Unbinds from a directory session.
ds_version()	Negotiates features of the interface and service.

The XDS Negotiation Sequence

The interface has an initialization and shutdown sequence that permits the negotiation of optional features. This involves the **ds_initialize()**, **ds_version()**, and **ds_shutdown()** functions.

Every application program must first call **ds_initialize()**, which returns a workspace. The workspace can be extended to support the optional Basic Directory Contents Package (see Chapter 12, “Basic Directory Contents Package” on page 275) or the Global Directory Service Package, or the MHS Directory User Package. These packages are identified by means of OSI Object Identifiers, and these Object Identifiers are supplied to **ds_version()** to incorporate the extensions into the workspace.

After a workspace with the required features is negotiated in this way, the application can use the workspace as required. It can create and manipulate OM objects using the OM functions, and can start one or more directory sessions using **ds_bind()**.

After completing its tasks, terminating all its directory sessions using **ds_unbind()**, and releasing all its OM objects using **om_delete()**, the application needs to ensure that resources associated with the interface are freed by calling **ds_shutdown()**.

The session Parameter

A session identifies the DUA and the suite of DSAs to which a particular directory operation is sent. It contains some **DirectoryBindArguments**, such as the distinguished name of the requestor. The *session* parameter is passed as the first parameter to most interface functions.

A session is described by an OM object of OM class **DS_C_SESSION**. After it is created, appropriate parameter values can be set using the OM functions. A directory session then starts with **ds_bind()** and later terminates with **ds_unbind()**. A session with default parameters can be started by passing the constant **DS_DEFAULT_SESSION** as the **DS_C_SESSION** parameter to **ds_bind()**.

The **ds_bind()** function must be called before **DS_C_SESSION** can be used as a parameter to any other function in this interface. After **ds_unbind()** is called, **ds_bind()** must be called again if another session is to be started.

The interface supports multiple concurrent sessions so that an application implemented as a single process, such as a server in a client/server model, can interact with the directory using several identities, and a process can interact directly and concurrently with different parts of the directory.

Details of the OM Class **DS_C_SESSION** are given in Chapter 11, “XDS Class Definitions” on page 241.

The context Parameter

The context defines the characteristics of the directory interaction that are specific to a particular directory operation; nevertheless, the same characteristics are often used for many operations. Because these parameters are assumed to be relatively static for a given directory user during a particular directory interaction, these parameters are collected into an OM object of OM class **DS_C_CONTEXT**, which is supplied as the second parameter of each Directory Service request. This reduces the number of parameters passed to each function.

The context includes many administrative details, such as the **CommonArguments** defined in the Abstract Service, which affect the processing of each directory operation. These include a number of **ServiceControls**, which allow control over some aspects of the service. The **ServiceControls** include options such as **preferChaining**, **chainingProhibited**, **localScope**, **dontUseCopy**, and **dontDereferenceAliases**, together with **priority**, **timeLimit**, **sizeLimit**, and **scopeOfReferral**. Each of these is mapped onto an OM attribute in the context. (See Chapter 11, “XDS Class Definitions” on page 241 for more information.)

The effect of passing the *context* parameter is as if its contents were passed as a group of additional arguments for every function call. The value of each component of the context is determined when the interface function is called, and remains fixed throughout the operation.

All OM attributes in the class **DS_C_CONTEXT** have default values, some of which are administered locally. The constant **DS_DEFAULT_CONTEXT** can be passed as the value of the **DS_C_CONTEXT** parameter to the interface functions, and has the same effect as a context OM object created with default values. The context must be a private object, unless it is **DS_DEFAULT_CONTEXT**.

See Chapter 11, “XDS Class Definitions” on page 241 for detailed specifications of the OM class **DS_C_CONTEXT**.

The XDS Function Arguments

The Abstract Service defines specific parameters for each operation. These are mapped onto corresponding parameters to each interface function, which are also called input parameters. Although each service has different arguments, some specific arguments recur in several operations and these are briefly introduced here. (For complete details of these parameters, see Chapter 11, “XDS Class Definitions” on page 241.)

All parameters that are OM objects can generally be supplied to the interface functions as public objects (that is, descriptor lists) or as private objects. Private objects must be created in the workspace that is returned by **ds_initialize()**. In some cases, constants can be supplied instead of OM objects.

Note: Wherever a function can accept an instance of a particular OM class as the value of a parameter, it also accepts an instance of any subclass of the OM class. For example, most functions have a

name parameter, which accepts values of OM class **DS_C_NAME**. It is always acceptable to supply an instance of the subclass **DS_C_DS_DN** as the value of the parameter.

Attribute and Attribute Value Assertion

Each directory attribute is represented in the interface by an OM object of OM class **DS_C_ATTRIBUTE**. The type of the directory attribute is represented by an OM attribute, **DS_ATTRIBUTE_TYPE**, within the OM object. The values of the directory attribute are expressed as the values of the OM attribute **DS_ATTRIBUTE_VALUES**.

The representation of the attribute value depends on the attribute type and is determined as indicated in the following list. The list describes the way in which an application program must supply values to the interface, for example, in the *changes* parameter to **ds_modify_entry()**. The interface follows the same rules when returning attribute values to the application, for example, in the **ds_read()** result.

- The first possibility is that the attribute type and the representation of the corresponding values can be defined in a package (for example, the selected attribute types from the standards that are defined in Chapter 12, “Basic Directory Contents Package” on page 275). In this case, attribute values are represented as specified. Additional directory attribute types and their OM representations are defined by the Global Directory Service Package.
- If the attribute type is not known and the value is an ASN.1 simple type, such as **IntegerType**, the representation is the corresponding type specified in Chapter 18, “XOM Service Interface” on page 331.
- If the attribute type is not known and the value is an ASN.1 structured type, the value is represented in the Basic Encoding Rules (BER) with OM syntax String(**OM_S_ENCODING**).

Note: The distinguished encoding specified in the Standards (see Clause 8.7 of *The Directory: Authentication Framework*, ISO 9594-8, CCITT X.500) must be used if the request is to be signed.

Where attribute values have OM syntax String(*), they can be long, segmented strings, and the functions **om_read()** and **om_write()** need to be used to access them.

An **Attribute Value Assertion (AVA)** is an assertion about the value of an attribute of an entry, and can be **OM_TRUE**, **OM_FALSE**, or undefined. It consists of an attribute type and a single value. In general, the AVA is **OM_TRUE** if one of the values of the given attribute in the entry matches the given value. An AVA is represented in the interface by an instance of OM class **DS_C_AVA**, which is a subclass of **DS_C_ATTRIBUTE** and can only have one value.

Information used by **ds_add_entry()** to construct a new directory entry is represented by an OM object of OM class **DS_C_ATTRIBUTE_LIST**, which contains a single multivalued OM attribute whose values are OM objects of OM class **DS_C_ATTRIBUTE**.

The Entry-Information-Selection Parameter

The *selection* parameter of the **ds_read()** and **ds_search()** operations tailors its results to obtain just part of the required entry. Information on all attributes, no attributes, or a specific group of attributes can be chosen. Attribute types are always returned, but the attribute values are not necessarily returned.

The value of the parameter is an instance of OM class **DS_C_ENTRY_INFORMATION_SELECTION**, but one of the constants in the following list can be used in simple cases:

- To verify the existence of an entry for the purported name, use the constant **DS_SELECT_NO_ATTRIBUTES**.
- To return just the types of all attributes, use the constant **DS_SELECT_ALL_TYPES**.

- To return the types and values of all attributes, use the constant **DS_SELECT_ALL_TYPES_AND_VALUES**.

To choose a particular set of attributes, create a new instance of the OM class **DS_C_ENTRY_INFORMATION_SELECTION** and set the appropriate OM attribute values using the OM functions.

The name Parameter

Most operations take a *name* parameter to specify the target of the operation. The name is represented by an instance of one of the subclasses of the OM class **DS_C_NAME**. The DCE XDS API defines the subclass **DS_C_DS_DN** to represent distinguished names and other names.

For directory interrogations, any aliases in the name are dereferenced, unless prohibited by the **DS_DONT_DEREFERENCE_ALIASES** service control. However, for modify operations, this service control is ignored if set, and aliases are never dereferenced.

RDNs are represented by an instance of one of the subclasses of the OM class **DS_C_RELATIVE_NAME**. The DCE XDS API defines the subclass **DS_C_DS_RDN** to represent RDNs.

XDS Function Call Results

All XDS functions return a **DS_status**, which is the C function result; most return data in an *invoke-id* parameter and the *result* parameter. The *invoke-id* and *result* values are returned using pointers that are supplied as parameters of the C function. These three types of function results are introduced in the following subsections.

All OM objects returned by interface functions (results and status) are private objects in the workspace returned by **ds_initialize()**.

The invoke-id Parameter

All interface functions that invoke a Directory Service operation return an *invoke-id* parameter, which is an integer that identifies the particular invocation of an operation. Because asynchronous operations are not supported, the *invoke-id* return value is no longer relevant for operations. DCE application programmers must still supply this parameter as described in the *z/OS DCE Application Development Reference*, but should ignore the value returned.

The result Parameter

Directory Service interrogation operations return a *result* value only if they succeed. All errors from these operations, including Directory Access Protocol (DAP) errors, are reported in **DS_status** (see “The DS_status Return Value” on page 239), as are errors from all other operations.

The result of an interrogation is returned in a private object whose OM class is appropriate to the particular operation. The format of directory operation results is driven by the Abstract Service. To simplify processing, the result of a single operation is returned in a single OM object, which corresponds to the abstract result defined in the standards. The components of the result of an operation are represented by OM attributes in the operation’s result object. All information contained in the Abstract Service result is made available to the application program. The result is inspected using the functions provided in the Object Management API, (**om_get()**).

Only the interrogation operations produce results, and each type of interrogation has a specific OM class of OM object for its result. These OM classes are as follows (see Chapter 11, “XDS Class Definitions” on page 241 for their definitions):

- **DS_C_COMPARE_RESULT**
- **DS_C_LIST_RESULT**
- **DS_C_READ_RESULT**
- **DS_C_SEARCH_RESULT.**

The results of the different operations share several common components, including the **CommonResults** defined in the standards (see *The Directory: Abstract Service Definition*, ISO 9594-3, CCITT X.511) by inheriting OM attributes from the superclass *DS_C_COMMON_RESULTS*. An additional common component is the full DN of the target object, after all aliases are dereferenced.

The actual OM class of the result can always be a subclass of that named to allow flexibility for extensions. Thus, **om_instance()** always needs to be used when testing the OM class.

Any attribute values in the result are represented as discussed in “Attribute and Attribute Value Assertion” on page 237.

The DS_status Return Value

Every interface function returns a **DS_status** value, which is either the constant **DS_SUCCESS** or an error. Errors are represented by private objects whose OM class is a subclass of **DS_C_ERROR**. Details of all errors are contained in “XDS Errors” on page 241.

Other results of functions are not valid unless the status result has the value **DS_SUCCESS**.

Synchronous Operations

Because asynchronous use of the interface is not supported, the value of the **DS_ASYNCHRONOUS** OM attribute in **DS_C_CONTEXT** is always **OM_FALSE**, causing all operations to be synchronous.

In synchronous mode, all functions wait until the operation is completed before returning. The thread of control is blocked within the interface after calling a function, and it can use the result immediately after the function returns.

Implementations define a limit on the number of asynchronous operations that can be outstanding at any one time on any one session. The limit is given by the implementation-defined constant **DS_MAX_OUTSTANDING_OPERATIONS**. It always has the value 0 (zero) because asynchronous operations are not supported.

All errors occurring during a synchronous request are reported when the function returns. (See “XDS Errors” on page 241 for complete details of error handling.)

Security and XDS

The X/Open XDS specifications do not define a security interface because this can put constraints on security features of existing directory implementations.

DCE GDS provides security by means of passwords. This is achieved at the XDS API level through a new **DSX_C_GDS_SESSION** session object with an OM **DSX_PASSWORD** attribute. The GDS DSA verifies this password for each directory operation.

Other Features of the XDS Interface

The following subsections describe these features of the interface:

- Automatic Connection Management
- Automatic Continuation and Referral Handling

Automatic Connection Management

An implementation can provide automatic management of the association or connection between the user and the Directory Service, making and releasing connections at its discretion.

The DCE XDS implementation does not support automatic connection management. A DSA connection is established when **ds_bind()** is called and released when **ds_unbind()** is called.

Automatic Continuation and Referral Handling

The interface provides automatic handling of continuation references and referrals in order to reduce the burden on application programs. These facilities can be inhibited to meet special needs.

A continuation reference describes how the performance of all or part of an operation can be continued at a different DSA or DSAs. A single continuation reference, returned as the entire response to an operation, is called a referral and is classified as an error. One or more continuation references can also be returned as part of **DS_PARTIAL_OUTCOME_QUAL** returned from a **ds_list()** or **ds_search()** operation.

A DSA returns a referral if it has administrative, operational, or technical reasons for preferring not to chain. It can return a referral if **DS_CHAINING_PROHIBITED** is set in the **DS_C_CONTEXT**, or instead it can report a service error (**DS_E_CHAINING_REQUIRED**) in this case.

By default, the implementation uses any continuation references it receives to try to contact the other DSA or DSAs, and so continues the operation, whenever practical. It only returns the result, or an error, to the application after it has made this attempt. Note that continuation references can still be returned to the application, if the relevant DSA cannot be contacted, for example.

The default behavior is the simplest for most applications, but if necessary the application can cause all continuation references to be returned to it. It does this by setting the value of the OM attribute **DS_AUTOMATIC_CONTINUATION** in the **DS_C_CONTEXT** to **OM_FALSE**.

Chapter 11. XDS Class Definitions

When referring to classes and attributes in the Directory Service, the chapters on GDS programming (Chapter 4, “GDS API: Concepts and Overview” on page 77 to Chapter 7, “Example Application Programs” on page 159) make a clear distinction between OM classes and directory classes, and between OM attributes and directory attributes. The former is a construct of the closely associated Object Management interface, while the latter is a construct of the *Directory Service* to which the interface provides access. The terms *object class* and *attribute* denote the directory constructs, while the phrases *OM class* and *OM attribute* denote the Object Management constructs.

Introduction to OM Classes

This chapter defines the OM classes that constitute the Directory Service Package, in alphabetical order. This package incorporates the OM classes for the errors which may be returned at the XDS interface. The object identifier associated with this package is **{iso(1) identified-organization(3) icd-ecma(0012) member-company(2) dec(1011) xopen(28) dsp(0)}** with the following encoding:

```
\x2B\x0C\x02\x87\x73\x1C\x00
```

This object identifier is represented by the constant **DS_SERVICE_PKG**.

The Object Management notation is briefly described in the following text. See Chapter 17, “Information Syntaxes” on page 325 through Chapter 19, “Object Management Package” on page 347 for more information on Object Management.

Each OM subclass is described in a separate section, which identifies the OM attributes specific to that subclass. The OM classes and OM attributes for each OM class are listed in alphabetical order. The OM attributes that can be found in an instance of an OM class are those OM attributes specific to that OM class, as well as those inherited from each of its superclasses. The OM class-specific OM attributes are defined in a table. The table indicates the name of each OM attribute, the syntax of each of its values, any restrictions upon the length (in bits, octets or bytes, or characters) of each value, any restrictions upon the number of values, and the value, if any, **om_create()** supplies.

The constants that represent the OM classes and OM attributes in the C binding are defined in the **xds.h** header file (listed in the *z/OS DCE Application Development Reference*).

XDS Errors

Errors are reported to the application program by means of **DS_status**, which is a result of most functions. (It is *the* function result in the C language binding for most functions.) A function that is completed successfully returns the value **DS_SUCCESS**, whereas one that is not successful returns an error. The error is a private object containing details of the problem that occurred. The error constant **DS_NO_WORKSPACE** can be returned by all Directory Service functions except **ds_initialize()** and **ds_shutdown()**. **DS_NO_WORKSPACE** is returned if **ds_initialize()** is not invoked before calling any other Directory Service function.

Errors are classified into ten OM classes. The standards (see *The Directory: Abstract Service Definition*, ISO 9594-3, CCITT X.511) classify errors into eight different groups, as follows:

- Abandoned
- Abandon Failed
- Attribute Error

- Name Error
- Referral
- Security Error
- Service Error
- Update Error

The Directory Service interface never returns an Abandoned error. The interface also defines three more kinds of errors: **DS_C_LIBRARY_ERROR**, **DS_C_COMMUNICATIONS_ERROR**, and **DS_C_SYSTEM_ERROR**. Each of these kinds of errors is represented by an OM class. These OM classes are detailed in subsequent sections of this chapter. All of them inherit the OM attribute **DS_PROBLEM** from their superclass *DS_C_ERROR*, which is described in this chapter. The error OM classes defined in this chapter are part of the Directory Service Package.

The **ds_bind()** operation returns a Security Error or a Service Error. All other operations can also return the same errors as **ds_bind()**. Such errors can arise in the course of following an automatic referral list.

DS_C_REFERRAL is not a real error, and it is not a subclass of *DS_C_ERROR*, although it is reported in the same way as a **DS_status** result. A **DS_C_ATTRIBUTE_ERROR**, also not a subclass of *DS_C_ERROR*, is special because it can report several problems at once. Each one is reported in a **DS_C_ATTRIBUTE_PROBLEM**, which is a subclass of *DS_C_ERROR*.

Note: The following errors referenced in this chapter do not apply to CDS:

- **DS_E_AFFECTS_MULTIPLE_DSAS**
- **DS_E_ATTRIBUTE_OR_VALUE_EXISTS**
- **DS_E_BAD_CLASS**
- **DS_E_BAD_WORKSPACE**
- **DS_E_INAPPROP_AUTHENTICATION**
- **DS_E_INVALID_REF**
- **DS_E_INVALID_SIGNATURE**
- **DS_E_MISSING_TYPE**
- **DS_E_MIXED SYNCHRONOUS**
- **DS_E_NO_INFO**
- **DS_E_NO_SUCH_OPERATION**
- **DS_E_NOT_SUPPORTED**
- **DS_E_OBJECT_CLASS_MOD_PROHIB**
- **DS_E_PROTECTION_REQUIRED**
- **DS_E_TOO_LATE**
- **DS_E_TOO_MANY_OPERATIONS**
- **DS_E_UNABLE_TO_PROCEED**
- **DS_E_UNAVAILABLE_CRIT_EXT**

OM Class Hierarchy

This section shows the hierarchical organization of the OM classes defined in this chapter and which OM classes inherit additional OM attributes from their superclasses. In the following list, subclassification is indicated by indentation, and the names of abstract classes are in italics. Thus, for example, the concrete class **DS_C_PRESENTATION_ADDRESS** is an immediate subclass of the abstract class *DS_C_ADDRESS*, which, in turn, is an immediate subclass of the abstract class *OM_C_OBJECT*. (*OM_C_OBJECT* is defined in Chapter 5, “XOM Programming” on page 97.)

Note: In the following list of OM classes, **DS_C_FILTER**, **DS_C_SEARCH_INFO** and **DS_SEARCH_RESULT** are GDS-specific, and therefore not supported.

OM_C_OBJECT

- **DS_C_ACCESS_POINT**
- *DS_C_ADDRESS*
 - **DS_C_PRESENTATION_ADDRESS**
- **DS_C_ATTRIBUTE**
 - **DS_C_AVA**
 - **DS_C_ENTRY_MOD**
 - **DS_C_FILTER_ITEM**
- **DS_C_ATTRIBUTE_ERROR**
- **DS_C_ATTRIBUTE_LIST**
 - **DS_C_ENTRY_INFO**
- *DS_C_COMMON_RESULTS*
 - **DS_C_COMPARE_RESULT**
 - **DS_C_LIST_INFO**
 - **DS_C_READ_RESULT**
 - **DS_C_SEARCH_INFO**
- **DS_C_CONTEXT**
- **DS_C_CONTINUATION_REF**
 - **DS_C_REFERRAL**
- **DS_C_ENTRY_INFO_SELECTION**
- **DS_C_ENTRY_MOD_LIST**
- *DS_C_ERROR*
 - **DS_C_ABANDON_FAILED**
 - **DS_C_ATTRIBUTE_PROBLEM**
 - **DS_C_COMMUNICATIONS_ERROR**
 - **DS_C_LIBRARY_ERROR**
 - **DS_C_NAME_ERROR**
 - **DS_C_SECURITY_ERROR**
 - **DS_C_SERVICE_ERROR**
 - **DS_C_SYSTEM_ERROR**
 - **DS_C_UPDATE_ERROR**
- **DS_C_EXT**
- **DS_C_FILTER**
- **DS_C_LIST_INFO_ITEM**
- **DS_C_LIST_RESULT**
- *DS_C_NAME*
 - **DS_C_DS_DN**
- **DS_C_OPERATION_PROGRESS**

- **DS_C_PARTIAL_OUTCOME_QUAL**
- *DS_C_RELATIVE_NAME*
 - **DS_C_DS_RDN**
- **DS_C_SEARCH_RESULT**
- **DS_C_SESSION**

None of the classes in the preceding list are encodable using **om_encode()** and **om_decode()**. The application is not permitted to create or modify instances of some OM classes because these OM classes are only returned by the interface and never supplied to it. These OM classes are as follows:

DS_C_ACCESS_POINT
DS_C_ATTRIBUTE_ERROR
DS_C_COMPARE_RESULT
DS_C_CONTINUATION_REF
All subclasses of *DS_C_ERROR*
DS_C_LIST_INFO
DS_C_LIST_INFO_ITEM
DS_C_LIST_RESULT
DS_C_OPERATION_PROGRESS
DS_C_PARTIAL_OUTCOME_QUAL
DS_C_READ_RESULT
DS_C_REFERRAL
DS_C_SEARCH_INFO
DS_C_SEARCH_RESULT

DS_C_ABANDON_FAILED

An instance of OM class **DS_C_ABANDON_FAILED** reports a problem encountered during an attempt to abandon an operation.

An application is not permitted to create or modify instances of this OM class. An instance of this OM class has the OM attributes of its superclasses, *OM_C_OBJECT* and *DS_C_ERROR*, and no additional OM attributes.

The OM attribute **DS_PROBLEM**, which is inherited from the superclass, *DS_C_ERROR*, identifies the problem. Its value is one of the following:

- **DS_E_CANNOT_ABANDON**

An attempt is made to abandon an operation that is prohibited, or the abandon action cannot be performed.
- **DS_E_NO_SUCH_OPERATION**

The Directory Service has no knowledge of the operation that is to be abandoned.
- **DS_E_TOO_LATE**

The operation is already completed, either successfully or erroneously.

The Directory Abandon operation is not supported by the DCE. Thus, a **ds_abandon()** XDS call always returns a **DS_E_TOO_LATE** error for the **DS_C_ABANDON_FAILED** OM class.

DS_C_ACCESS_POINT

An instance of OM class **DS_C_ACCESS_POINT** identifies a particular point at which a DSA can be accessed.

An application is not permitted to create or modify instances of this OM class. An instance of this OM class has the OM attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attributes listed in Table 21.

Table 21. OM Attributes of *DS_C_ACCESS_POINT*

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_ADDRESS	Object(<i>DS_C_ADDRESS</i>)	-	1	-
DS_AE_TITLE	Object(<i>DS_C_NAME</i>)	-	1	-

- **DS_ADDRESS**

This attribute indicates the address of the DSA to be used when communicating with it.

- **DS_AE_TITLE**

This attribute indicates the name of the DSA.

DS_C_ADDRESS

The OM class *DS_C_ADDRESS* represents the address of a particular entity or service, such as a DSA.

It is an abstract class that has the OM attributes of its superclass, *OM_C_OBJECT*, and no other OM attributes.

An address is an unambiguous name, label, or number that identifies the location of the entity or service. All addresses are represented as instances of some subclass of this OM class. The only subclass defined by the DCE XDS API is **DS_C_PRESENTATION_ADDRESS**, which is the presentation address of an OSI application entity used for OSI communications with this subclass.

DS_C_ATTRIBUTE

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

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

Table 22. OM Attributes of *DS_C_ATTRIBUTE*

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_ATTRIBUTE_TYPE	String(<i>OM_S_OBJECT_IDENTIFIER_STRING</i>)	-	1	-
DS_ATTRIBUTE_VALUES	Any	-	0 or more	-

- **DS_ATTRIBUTE_TYPE**

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

- **DS_ATTRIBUTE_VALUES**

The attribute values. The OM value syntax and the number of values allowed for this OM attribute are determined by the value of the **DS_ATTRIBUTE_TYPE** OM attribute in accordance with the rules in “Attribute and Attribute Value Assertion” on page 237.

If the values of this OM attribute have the syntax String(*), the strings can be long and segmented. For this reason, **om_read()** and **om_write()** need to be used to access all String(*) values.

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

DS_C_ATTRIBUTE_ERROR

An instance of OM class **DS_C_ATTRIBUTE_ERROR** reports an attribute-related Directory Service error.

An application is not permitted to create or modify instances of this OM class. An instance of this OM class has the OM attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attributes listed in Table 23.

Table 23. OM Attributes of DS_C_ATTRIBUTE_ERROR

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_OBJECT_NAME	Object(<i>DS_C_NAME</i>)	-	1	-
DS_PROBLEMS	Object(DS_C_ATTRIBUTE_PROBLEM)	-	1 or more	-

- **DS_OBJECT_NAME**

This attribute contains the name of the directory entry to which the operation is applied when the failure occurs.

- **DS_PROBLEMS**

This attribute documents the attribute-related problems encountered. A **DS_C_ATTRIBUTE_ERROR** can report several problems at once. All problems are related to the preceding object.

DS_C_ATTRIBUTE_LIST

An instance of OM class **DS_C_ATTRIBUTE_LIST** is a list of directory attributes.

An instance of this OM class has the OM attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attributes listed in Table 24.

Table 24. OM Attributes of DS_C_ATTRIBUTE_LIST

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_ATTRIBUTES	Object(DS_C_ATTRIBUTE)	-	0 or more	-

- **DS_ATTRIBUTES**

The attributes that constitute a new object's directory entry, or those selected from an existing entry.

DS_C_ATTRIBUTE_PROBLEM

An instance of OM class **DS_C_ATTRIBUTE_PROBLEM** documents one attribute-related problem encountered while performing an operation as requested on a particular occasion.

An application is not permitted to create or modify instances of this OM class. An instance of this OM class has the OM attributes of its superclasses, *OM_C_OBJECT* and *DS_C_ERROR* in addition to the OM attributes listed in Table 25.

Table 25. OM Attributes of *DS_C_ATTRIBUTE_PROBLEM*

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_ATTRIBUTE_TYPE	String(OM_S_OBJECT_IDENTIFIER_STRING)	-	1	-
DS_ATTRIBUTE_VALUE	Any	-	0 or 1	-

- **DS_ATTRIBUTE_TYPE**

This attribute identifies the type of attribute with which the problem is associated.

- **DS_ATTRIBUTE_VALUE**

The attribute specifies the attribute value with which the problem is associated. Its syntax is determined by the value of **DS_ATTRIBUTE_TYPE**. This OM attribute is present if it is necessary to avoid ambiguity.

The OM attribute **DS_PROBLEM**, which is inherited from the superclass, *DS_C_ERROR*, identifies the problem. Its value is one of the following:

- **DS_E_ATTRIBUTE_OR_VALUE_EXISTS**

An attempt is made to add an attribute or value that is already present in the directory entry in question.

- **DS_E_CONSTRAINT_VIOLATION**

The attribute or attribute value does not conform to the constraints imposed by the standards (see *The Directory: Models*, ISO 9594-2, CCITT X.501) or by the attribute definition; for example, the value exceeds its upper bound.

- **DS_E_INAPPROP_MATCHING**

An attempt is made to use a matching rule that is not defined for the attribute type.

- **DS_E_INVALID_ATTRIBUTE_SYNTAX**

A value presented as an argument does not conform to the attribute syntax of the attribute type.

- **DS_E_NO_SUCH_ATTRIBUTE_OR_VALUE**

The specified attribute or value is not found in the directory entry in question. This error is only reported by a **ds_read()** or **ds_search()** operation if an explicit list of attributes is specified by the *selection* argument, but none of them are present in the entry.

- **DS_E_UNDEFINED_ATTRIBUTE_TYPE**

The attribute type, which is supplied as an argument to **ds_add_entry()** or **ds_modify_entry()**, is undefined.

DS_C_AVA

An instance of OM class **DS_C_AVA** (attribute value assertion) is a proposition concerning the values of a directory entry.

An instance of this OM class has the OM attributes of its superclasses, *OM_C_OBJECT* and **DS_C_ATTRIBUTE**, and no other OM attributes. An additional restriction on this OM class is that there must be exactly one value of the OM attribute **DS_ATTRIBUTE_VALUES**. The **DS_ATTRIBUTE_TYPE** remains single valued. The OM value syntax of **DS_ATTRIBUTE_VALUES** must conform to the rules outlined in “Attribute and Attribute Value Assertion” on page 237.

DS_C_COMMON_RESULTS

The OM class *DS_C_COMMON_RESULTS* comprises results that are returned by, and are common to, the directory interrogation operations.

It is an abstract OM class, which has the OM attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attributes listed in Table 26.

Table 26. OM Attributes of *DS_C_COMMON_RESULTS*

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_ALIAS_DEREFERENCED	OM_S_BOOLEAN	-	1	-
DS_PERFORMER	Object(<i>DS_C_NAME</i>)	-	0-1	-

- **DS_ALIAS_DEREFERENCED**

This attribute indicates whether the name of the target object that is passed as a function argument includes an alias that is dereferenced to determine the distinguished name (DN).

- **DS_PERFORMER**

When present, this attribute gives the DN of the performer of a particular operation. It can be present when the result is signed, and it holds the name of the DSA that signed the result. The DCE Directory Service does not support the optional feature of signed results; therefore, this OM attribute is never present.

DS_C_COMMUNICATIONS_ERROR

An instance of OM class **DS_C_COMMUNICATIONS_ERROR** reports an error occurring in the other OSI services supporting the Directory Service.

An application is not permitted to create or modify instances of this OM class. An instance of this OM class has the OM attributes of its superclasses, *OM_C_OBJECT* and *DS_C_ERROR*, and no additional OM attributes.

Communications errors include those arising in remote operation, association control, presentation, session, and transport.

The OM attribute **DS_PROBLEM**, which is inherited from the superclass *DS_C_ERROR*, identifies the problem. Its value is **DS_E_COMMUNICATIONS_PROBLEM**.

DS_C_COMPARE_RESULT

An instance of OM class **DS_C_COMPARE_RESULT** comprises the results of a successful call to **ds_compare()**.

An application is not permitted to create or modify instances of this OM class. An instance of this OM class has the OM attributes of its superclasses, *OM_C_OBJECT* and *DS_C_COMMON_RESULTS*, in addition to the OM attributes listed in Table 27.

Table 27. OM Attributes of *DS_C_COMPARE_RESULT*

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_FROM_ENTRY	OM_S_BOOLEAN	-	1	-
DS_MATCHED	OM_S_BOOLEAN	-	1	-
DS_OBJECT_NAME	Object(<i>DS_C_NAME</i>)	-	0 or 1	-

- **DS_FROM_ENTRY**

This attribute indicates whether the assertion is tested against the specified object's entry, rather than a copy of the entry.

- **DS_MATCHED**

Indicates whether the assertion specified as an argument returns the value **OM_TRUE**. It takes the value **OM_TRUE** if the values are compared and matched; otherwise, it takes the value **OM_FALSE**.

- **DS_OBJECT_NAME**

This attribute contains the distinguished name of the target object of the operation. It is present if the OM attribute **DS_ALIAS_DEREFERENCED**, inherited from the superclass, *DS_C_COMMON_RESULTS*, is **OM_TRUE**.

DS_C_CONTEXT

An instance of OM class **DS_C_CONTEXT** comprises per-operation arguments that are accepted by most of the interface functions.

An instance of this OM class has the OM attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attributes listed in Table 28.

Table 28 (Page 1 of 2). OM Attributes of *DS_C_CONTEXT*

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
Common Arguments:				
DS_EXT	Object(DS_C_EXT)	-	0 or more	-
DS_OPERATION_PROGRESS	Object(DS_C_OPERATION_PROGRESS)	-	1	DS_OPERATION_NOT_STARTED

Table 28 (Page 2 of 2). OM Attributes of DS_C_CONTEXT

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_ALIASED_RDNS	OM_S_INTEGER	-	0 or 1	0
Service Controls:				
DS_CHAINING_PROHIB	OM_S_BOOLEAN	-	1	OM_TRUE
DS_DONT_DEREFERENCE_ALIASES	OM_S_BOOLEAN	-	1	OM_FALSE
DS_DONT_USE_COPY	OM_S_BOOLEAN	-	1	OM_TRUE
DS_LOCAL_SCOPE	OM_S_BOOLEAN	-	1	OM_FALSE
DS_PREFER_CHAINING	OM_S_BOOLEAN	-	1	OM_FALSE
DS_PRIORITY	Enum(DS_priority)	-	1	DS_MEDIUM
DS_SCOPE_OF_REFERRAL	Enum(DS_Scope_Of_Referral)	-	0 or 1	-
DS_SIZE_LIMIT	OM_S_INTEGER	-	0 or 1	-
DS_TIME_LIMIT	OM_S_INTEGER	-	0 or 1	-
Local Controls:				
DS_ASYNCHRONOUS	OM_S_BOOLEAN	-	1	OM_FALSE
DS_AUTOMATIC_CONTINUATION	OM_S_BOOLEAN	-	1	OM_TRUE

The context gathers several arguments passed to interface functions, which are assumed to be relatively static for a given directory user during a particular directory interaction. The context is passed as an argument to each function that interrogates or updates the directory. Although it is generally assumed that the context is changed infrequently, the value of each argument can be changed between every operation if required. The **DS_ASYNCHRONOUS** argument must not be changed. Each argument is represented by one of the OM attributes of the **DS_C_CONTEXT** OM class.

The context contains the common arguments defined in the standards (see *The Directory: Abstract Service Definition*, ISO 9594-3, CCITT X.511), except that all security information is omitted for reasons discussed in “Security and XDS” on page 240. These are made up of a number of service controls explained in the following text. It also contains a number of arguments that provide local control over the interface.

The OM attributes of the **DS_C_CONTEXT** OM class are:

- Common Arguments

- **DS_EXT**

This attribute represents any future standardized extensions that need to be applied to the Directory Service operation. The DCE XDS implementation does not evaluate this optional OM attribute.

- **DS_OPERATION_PROGRESS**

This attribute represents the state that the Directory Service assumes at the start of the operation. This OM attribute normally takes its default value, which is the value **DS_OPERATION_NOT_STARTED** described in the **DS_C_OPERATION_PROGRESS** OM class definition.

- **DS_ALIASED_RDNS**

This attribute indicates to the Directory Service that the object component of the *operation* parameter is created by dereferencing of an alias on an earlier operation attempt. This value is set in the referral response of the previous operation.

- Service Controls

- **DS_CHAINING_PROHIB**

- This attribute indicates that chaining and other methods of distributing the request around the Directory Service are prohibited.

- **DS_DONT_DEREFERENCE_ALIASES**

- This attribute indicates that any alias used to identify the target entry of an operation is not dereferenced. With this attribute, you can interrogate alias entries (aliases are never dereferenced during updates).

- **DS_DONT_USE_COPY**

- This attribute indicates that the request can only be satisfied by accessing directory entries, and not by using copies of entries. This includes both copies maintained in other DSAs by bilateral agreement, and copies cached locally.

- **DS_LOCAL_SCOPE**

- This attribute indicates that the directory request will be satisfied locally. The meaning of this option is configured by an administrator. This option typically restricts the request to a single DSA.

- **DS_PREFER_CHAINING**

- This attribute indicates that chaining is preferred to referrals when necessary. The Directory Service is not obliged to follow this preference, and can return a referral even if it is set.

- **DS_PRIORITY**

- This attribute indicates the priority, relative to other directory requests, according to which the Directory Service attempts to satisfy the request. This is not a guaranteed service since there is no queuing throughout the directory. Its value must be one of the following:

- **DS_LOW**

- **DS_MEDIUM**

- **DS_HIGH**

- **DS_SCOPE_OF_REFERRAL**

- This attribute indicates the part of the directory to which referrals are limited. This includes referral errors and partial outcome qualifiers. Its value must be one of the following:

- **DS_COUNTRY**, meaning DSAs within the country in which the request originates.

- **DS_DMD**, meaning DSAs within the DMD in which the request originates.

- DS_SCOPE_OF_REFERRAL** is an optional attribute. The lack of this attribute in a **DS_C_CONTEXT** object indicates that the scope is not limited.

- **DS_SIZE_LIMIT**

- If present, this attribute indicates the maximum number of objects about which **ds_list()** or **ds_search()** needs to return information. If this limit is exceeded, information is returned about exactly this number of objects. The objects that are chosen are not specified because this can depend on the timing of interactions between DSAs, among other reasons.

- **DS_TIME_LIMIT**

If present, this attribute indicates the maximum elapsed time, in seconds, within which the service needs to be provided (not the processing time devoted to the request). If this limit is reached, a service error (**DS_E_TIME_LIMIT_EXCEEDED**) is returned, except for the **ds_list()** or **ds_search()** operations, which return an arbitrary selection of the accumulated results.

- Local Controls

- **DS_ASYNCHRONOUS** (NOT SUPPORTED)

The interface operates only synchronously as detailed in “Synchronous Operations” on page 239. There is only one possible value:

- **OM_FALSE** indicates that the operation is performed sequentially (synchronously) with the application being blocked until a result or error is returned.

- **DS_AUTOMATIC_CONTINUATION**

This attribute indicates the requestor’s requirement for continuation reference handling, including referrals and those in partial outcome qualifiers. The value is one of the following:

- **OM_FALSE** indicates that the interface returns all continuation references to the application program.
 - **OM_TRUE** indicates that continuation references are automatically processed, and the subsequent results are returned to the application instead of the continuation references, whenever practical. This is a much simpler option than **OM_FALSE** unless the application has special requirements.

Note: Continuation references can still be returned to the application, if for example, the relevant DSA cannot be contacted.

Applications can assume that an object of OM class **DS_C_CONTEXT**, created with default values of all its OM attributes, works with all the interface functions. The **DS_DEFAULT_CONTEXT** constant can be used as an argument to interface functions instead of creating an OM object with default values.

DS_C_CONTINUATION_REF

An instance of OM class **DS_C_CONTINUATION_REF** comprises the information that enables a partially completed directory request to be continued, for example, following a referral.

An application is not permitted to create or modify instances of this OM class. An instance of this OM class has the OM attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attributes listed in Table 29.

Table 29. OM Attributes of DS_C_CONTINUATION_REF

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_ACCESS_POINTS	Object(DS_C_ACCESS_POINT)	-	1 or more	-
DS_ALIASED_RDNS	OM_S_INTEGER	-	1	-
DS_OPERATION_PROGRESS	Object(DS_C_OPERATION_PROGRESS)	-	1	-
DS_RDNS_RESOLVED	OM_S_INTEGER	-	0 or 1	-
DS_TARGET_OBJECT	Object(<i>DS_C_NAME</i>)	-	1	-

- **DS_ACCESS_POINTS**

This attribute indicates the names and presentation addresses of the DSAs from where the directory request is continued.

- **DS_ALIASED_RDNS**

This attribute indicates how many (if any) of the RDNs in the target name are produced by dereferencing an alias. Its value is 0 (zero) if no aliases are dereferenced. This value needs to be used in the **DS_C_CONTEXT** of any continued operation.

- **DS_OPERATION_PROGRESS**

This attribute indicates the state at which the directory request must be continued. This value needs to be used in the **DS_C_CONTEXT** of any continued operation.

- **DS_RDNS_RESOLVED**

This attribute indicates the number of RDNs in the supplied object name that are resolved (using internal references), and not just assumed to be correct (using cross-references).

- **DS_TARGET_OBJECT**

This attribute indicates the name of the object upon which the continuation must focus.

DS_C_DS_DN

An instance of OM class **DS_C_DS_DN** represents a name of a directory object.

An instance of this OM class has the OM attributes of its superclasses, *OM_C_OBJECT* and *DS_C_NAME*, in addition to the OM attributes listed in Table 30.

Table 30. OM Attribute of DS_C_DS_DN

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_RDNS	Object(DS_C_DS_RDN)	-	0 or more	-

- **DS_RDNS**

This attribute indicates the sequence of RDNs that define the path through the DIT from its root to the object that the **DS_C_DS_DN** indicates. The **DS_C_DS_DN** of the root of the directory is the null name (no **DS_RDNS** values). The order of the values is significant; the first value is closest to the root, and the last value is the RDN of the object.

DS_C_DS_RDN

An instance of OM class **DS_C_DS_RDN** is a relative distinguished name. An RDN uniquely identifies an immediate subordinate of an object whose entry is displayed in the DIT.

An instance of this OM class has the OM attributes of its superclasses, *OM_C_OBJECT* and *DS_C_RELATIVE_NAME*, in addition to the OM attributes listed in Table 31.

Table 31. OM Attribute of DS_C_DS_RDN

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_AVAS	Object(DS_C_AVA)	-	1 or more	-

- **DS_AVAS**

This attribute indicates the **DS_AVAS** that are marked by the DIB as components of the object's RDN. The assertion is **OM_TRUE** of the object but not of any of its siblings, and the attribute type and value are displayed in the object's directory entry.

DS_C_ENTRY_INFO

An instance of OM class **DS_C_ENTRY_INFO** contains selected information from a single directory entry.

An instance of this OM class has the OM attributes of its superclasses, *OM_C_OBJECT* and **DS_C_ATTRIBUTE_LIST**, in addition to the OM attributes listed in Table 32.

Table 32. OM Attributes of DS_C_ENTRY_INFO

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_FROM_ENTRY	OM_S_BOOLEAN	-	1	-
DS_OBJECT_NAME	Object(DS_C_NAME)	-	1	-

The OM attribute **DS_ATTRIBUTES** is inherited from the superclass **DS_C_ATTRIBUTE_LIST**. It contains the information extracted from the directory entry of the target object. The type of each attribute requested and located is indicated in the list as are its values, if types and values are requested.

The OM class-specific OM attributes are as follows:

- **DS_FROM_ENTRY**

This attribute indicates whether the information is extracted from the specified object's entry, rather than from a copy of the entry.

- **DS_OBJECT_NAME**

This attribute contains the object's distinguished name.

DS_C_ENTRY_INFO_SELECTION

An instance of OM class **DS_C_ENTRY_INFO_SELECTION** identifies the information to be extracted from a directory entry.

An instance of this OM class has the OM attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attributes listed in Table 33.

Table 33. OM Attributes of DS_C_ENTRY_INFO_SELECTION

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_ALL_ATTRIBUTES	OM_S_BOOLEAN	-	1	OM_TRUE
DS_ATTRIBUTES_SELECTED	String(OM_S_OBJECT_IDENTIFIER_STRING)	-	0 or more	-
DS_INFO_TYPE	Enum(DS_Information_Type)	-	1	DS_TYPES_AND_VALUES

- **DS_ALL_ATTRIBUTES**

This attribute indicates which attributes are relevant. It can take one of the following values:

- **OM_FALSE** indicates that information is only requested on those attributes that are listed in the OM attribute **DS_ATTRIBUTES_SELECTED**.
- **OM_TRUE** indicates that information is requested on all attributes in the directory entry. Any values of the OM attribute **DS_ATTRIBUTES_SELECTED** are ignored in this case.

- **DS_ATTRIBUTES_SELECTED**

This attribute lists the types of attributes in the entry from which information will be extracted. The value of this OM attribute is only used if the value of **DS_ALL_ATTRIBUTES** is **OM_FALSE**. If an empty list is supplied, no attribute data is returned that can be used to verify the existence of an entry for a distinguished name.

- **DS_INFO_TYPE**

This attribute identifies what information will be extracted from each attribute identified. It must take one of the following values:

- **DS_TYPES_ONLY** indicates that only the attribute types of the selected attributes in the entry are returned.
- **DS_TYPES_AND_VALUES** indicates that both the attribute types and the attribute values of the selected attributes in the entry are returned.

DS_C_ENTRY_MOD

An instance of OM class **DS_C_ENTRY_MOD** describes a single modification to a specified attribute of a directory entry.

An instance of this OM class has the OM attributes of its superclasses, *OM_C_OBJECT* and **DS_C_ATTRIBUTE**, in addition to the OM attributes listed in Table 34.

Table 34. OM Attributes of DS_C_ENTRY_MOD

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_MOD_TYPE	Enum(DS_Modification_Type)	-	1	DS_ADD_ATTRIBUTE

The attribute type to be modified and the associated values are specified in the OM attributes **DS_ATTRIBUTE_TYPE** and **DS_ATTRIBUTE_VALUES** that are inherited from the **DS_C_ATTRIBUTE** superclass.

- **DS_MOD_TYPE**

This attribute identifies the type of modification. It must have one of the following values:

- **DS_ADD_ATTRIBUTE** indicates that the specified attribute is absent and is to be added with the specified values.
- **DS_ADD_VALUES** indicates that the specified attribute is present and that one or more specified values are to be added to it.
- **DS_REMOVE_ATTRIBUTE** indicates that the specified attribute is present and will be removed. Any values present in the OM attribute **DS_ATTRIBUTE_VALUES** are ignored.
- **DS_REMOVE_VALUES** indicates that the specified attribute is present and that one or more specified values will be removed from it.

DS_C_ENTRY_MOD_LIST

An instance of OM class **DS_C_ENTRY_MOD_LIST** comprises a sequence of changes to be made to a directory entry.

An instance of this OM class has the OM attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attributes listed in Table 35.

Table 35. OM Attributes of DS_C_ENTRY_MOD_LIST

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_CHANGES	Object(DS_C_ENTRY_MOD)	-	1 or more	-

- **DS_CHANGES**

This attribute identifies the modifications to be made (in the order specified) to the directory entry of the specified object.

DS_C_ERROR

The OM class *DS_C_ERROR* comprises the parameters common to all errors.

It is an abstract OM class with the OM attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attribute listed in Table 36.

Table 36. OM Attributes of DS_C_ERROR

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_PROBLEM	Enum(DS_Problem)	-	1	-

Details of errors are returned in an instance of a subclass of this OM class. Each such subclass represents a particular kind of error, and is one of the following:

- **DS_C_ABANDON_FAILED**
- **DS_C_ATTRIBUTE_PROBLEM**
- **DS_C_COMMUNICATIONS_ERROR**
- **DS_C_LIBRARY_ERROR**

- **DS_C_NAME_ERROR**
- **DS_C_SECURITY_ERROR**
- **DS_C_SERVICE_ERROR**
- **DS_C_SYSTEM_ERROR**
- **DS_C_UPDATE_ERROR**

A number of possible values for the **DS_PROBLEM** attribute are defined for these subclasses. DCE XDS does not return other values for error conditions described in this chapter. Information on system errors can be found in “DS_C_SYSTEM_ERROR” on page 272. Each of the following standard values for the **DS_PROBLEM** attribute is described under the relevant error OM class:

- **DS_E_ADMIN_LIMIT_EXCEEDED**
- **DS_E_AFFECTS_MULTIPLE_DSAS**
- **DS_E_ALIAS_DEREFERENCING_PROBLEM**
- **DS_E_ALIAS_PROBLEM**
- **DS_E_ATTRIBUTE_OR_VALUE_EXISTS**
- **DS_E_BAD_ARGUMENT**
- **DS_E_BAD_CLASS**
- **DS_E_BAD_CONTEXT**
- **DS_E_BAD_NAME**
- **DS_E_BAD_SESSION**
- **DS_E_BAD_WORKSPACE**
- **DS_E_BUSY**
- **DS_E_CANNOT_ABANDON**
- **DS_E_CHAINING_REQUIRED**
- **DS_E_COMMUNICATIONS_PROBLEM**
- **DS_E_CONSTRAINT_VIOLATION**
- **DS_E_DIT_ERROR**
- **DS_E_ENTRY_EXISTS**
- **DS_E_INAPPROP_AUTHENTICATION**
- **DS_E_INAPPROP_MATCHING**
- **DS_E_INSUFFICIENT_ACCESS_RIGHTS**
- **DS_E_INVALID_ATTRIBUTE_SYNTAX**
- **DS_E_INVALID_ATTRIBUTE_VALUE**
- **DS_E_INVALID_CREDENTIALS**
- **DS_E_INVALID_REF**
- **DS_E_INVALID_SIGNATURE**
- **DS_E_LOOP_DETECTED**
- **DS_E_MISCELLANEOUS**
- **DS_E_MISSING_TYPE**

- **DS_E_MIXED_SYNCHRONOUS**
- **DS_E_NAMING_VIOLATION**
- **DS_E_NO_INFO**
- **DS_E_NO_SUCH_ATTRIBUTE_OR_VALUE**
- **DS_E_NO_SUCH_OBJECT**
- **DS_E_NO_SUCH_OPERATION**
- **DS_E_NOT_ALLOWED_ON_NON_LEAF**
- **DS_E_NOT_ALLOWED_ON_RDN**
- **DS_E_NOT_SUPPORTED**
- **DS_E_OBJECT_CLASS_MOD_PROHIB**
- **DS_E_OBJECT_CLASS_VIOLATION**
- **DS_E_OUT_OF_SCOPE**
- **DS_E_PROTECTION_REQUIRED**
- **DS_E_TIME_LIMIT_EXCEEDED**
- **DS_E_TOO_LATE**
- **DS_E_TOO_MANY_OPERATIONS**
- **DS_E_TOO_MANY_SESSIONS**
- **DS_E_UNABLE_TO_PROCEED**
- **DS_E_UNAVAILABLE**
- **DS_E_UNAVAILABLE_CRIT_EXT**
- **DS_E_UNDEFINED_ATTRIBUTE_TYPE**
- **DS_E_UNWILLING_TO_PERFORM**

DS_C_EXT

An instance of OM class **DS_C_EXT** indicates a standardized extension to the Directory Service is outlined in post-1988 versions of the standards. Therefore, this OM class is not used by the XDS API and is only included for X/Open conformance purposes.

An instance of this OM class has the OM attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attributes listed in Table 37.

Table 37. OM Attributes of DS_C_EXT

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_CRIT	OM_S_BOOLEAN	-	1	OM_FALSE
DS_IDENT	OM_S_INTEGER	-	1	-
DS_ITEM_PARAMETERS	Any	-	1	-

- **DS_CRIT**

This attribute must have one of the following values:

- **OM_FALSE** indicates that the originator permits the operation to be performed even if the extension is not available.
 - **OM_TRUE** indicates that the originator mandates that the extended operation be performed. If the extended operation is not performed, an error is reported.
- **DS_IDENT**
This attribute identifies the service extension.
 - **DS_ITEM_PARAMETERS**
This OM attribute supplies the parameters of the extension. Its syntax is determined by the value of **DS_IDENT**.

DS_C_FILTER

Note: **DS_C_FILTER** is not useful for CDS applications. It is intended for use with **ds_search()**, which is not supported.

An instance of OM class **DS_C_FILTER** is used to select or reject an object on the basis of information in its directory entry. At any point, an attribute filter has a value relative to every object. The value is **OM_FALSE**, **OM_TRUE**, or undefined. The object is selected if, and only if, the filter's value is **OM_TRUE**.

An instance of this OM class has the OM attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attributes listed in Table 38.

Table 38. OM Attributes of DS_C_FILTER

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_FILTER_ITEMS	Object(DS_C_FILTER_ITEM)	-	0 or more	-
DS_FILTERS	Object(DS_C_FILTER)	-	0 or more	-
DS_FILTER_TYPE	Enum(DS_Filter_Type)	-	1	DS_AND

A *filter* is a collection of less elaborate filters and elementary **DS_FILTER_ITEMS** together with a Boolean operation. The filter value is undefined if, and only if, all the component **DS_FILTERS** and **DS_FILTER_ITEMS** are undefined. Otherwise, the filter has a Boolean value with respect to any directory entry, which can be determined by evaluating each of the nested components and combining their values using the Boolean operation. The components whose values are undefined are ignored.

- **DS_FILTER_ITEMS**
This attribute is a collection of assertions, each relating to just one attribute of a directory entry.
- **DS_FILTERS**
This attribute is a collection of simpler filters.
- **DS_FILTER_TYPE**
This attribute is the filter's type. It can have any of the following values:
 - **DS_AND** indicates that the filter is the logical conjunction of its components. The filter is **OM_TRUE** unless any of the nested filters or filter items is **OM_FALSE**. If there are no nested components, the filter is **OM_TRUE**.

- **DS_OR** indicates that the filter is the logical disjunction of its components. The filter is **OM_FALSE** unless any of the nested filters or filter items is **OM_TRUE**. If there are no nested components, the filter is **OM_FALSE**.
- **DS_NOT** indicates that the result of this filter is reversed. There must be exactly one nested filter or filter item. The filter is **OM_TRUE** if the enclosed filter or filter item is **OM_FALSE**, and is **OM_FALSE** if the enclosed filter or filter item is **OM_TRUE**.

DS_C_FILTER_ITEM

Note: **DS_C_FILTER_ITEMS** is not useful for CDS applications. It is intended for use with **ds_search()**, which is not supported.

An instance of OM class **DS_C_FILTER_ITEM** is a component of **DS_C_FILTER**. It is an assertion about the existence or values of a single attribute type in a directory entry.

An instance of this OM class has the OM attributes of its superclasses, *OM_C_OBJECT* and **DS_C_ATTRIBUTE**, in addition to the OM attributes listed in Table 39.

Table 39. OM Attributes of DS_C_FILTER_ITEM

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_FILTER_ITEM_TYPE	Enum(DS_Filter_Item_Type)	-	1	-
DS_FINAL_SUBSTRING	String(*)	1 or more	0 or 1	-
DS_INITIAL_SUBSTRING	String(*)	1 or more	0 or 1	-

Note: OM attributes **DS_ATTRIBUTE_TYPE** and **DS_ATTRIBUTE_VALUES** are inherited from the superclass **DS_C_ATTRIBUTE**.

The value of the filter item is undefined in the following cases:

- The **DS_ATTRIBUTE_TYPE** is not known.
- None of the **DS_ATTRIBUTE_VALUES** conform to the attribute syntax defined for that attribute type.
- The **DS_FILTER_ITEM_TYPE** uses a matching rule that is not defined for the attribute syntax.

Access control restrictions can also cause the value to be undefined.

- **DS_FILTER_ITEM_TYPE**

This attribute identifies the type of filter item and thus, the nature of the filter. The filter item can adopt any of the following values:

- **DS_APPROXIMATE_MATCH** indicates that the filter is **OM_TRUE**, if the directory entry contains at least one value of the specified type that is approximately equal to that specified (the meaning of *approximately equal* is implementation dependent). Otherwise, the filter is **OM_FALSE**.

Rules for approximate matching are defined locally. For example, an approximate match may take into account spelling variations or employ phonetic comparison rules. In the absence of any such capabilities, a DSA needs to treat an approximate match as a test for equality. DCE GDS supports phonetic comparisons. There must be exactly one value of the OM attribute **DS_ATTRIBUTE_VALUES**.

- **DS_EQUALITY** indicates that the filter is **OM_TRUE**, if the entry contains at least one value of the specified type that is equal to the value specified, according to the equality matching rule in force. Otherwise, the filter is **OM_FALSE**. There must be exactly one value of the OM attribute **DS_ATTRIBUTE_VALUES**.
- **DS_GREATER_OR_EQUAL** indicates that the filter item is **OM_TRUE** if, and only if, at least one value of the attribute is greater than or equal to the supplied value (using the appropriate ordering algorithm). There must be exactly one value of the OM attribute **DS_ATTRIBUTE_VALUES**.
- **DS_LESS_OR_EQUAL** indicates that the filter item is **OM_TRUE** if, and only if, at least one value of the attribute is less than or equal to the supplied value. There must be exactly one value of the OM attribute **DS_ATTRIBUTE_VALUES**.
- **DS_PRESENT** indicates that the filter is **OM_TRUE**, if the entry contains an attribute of the specified type. Otherwise, it is **OM_FALSE**. Any values of the OM attribute **DS_ATTRIBUTE_VALUES** are ignored.
- **DS_SUBSTRINGS** indicates that the filter is **OM_TRUE**, if the entry contains at least one value of the specified attribute type that contains all of the specified substrings in the given order. Otherwise, the filter is **OM_FALSE**.

Any number of substrings can be given as values of the OM attribute **DS_ATTRIBUTE_VALUES**. Similarly, no substrings can be specified as values of the OM attribute **DS_ATTRIBUTE_VALUES**. There can also be a substring in **DS_INITIAL_SUBSTRING** or **DS_FINAL_SUBSTRING**, or both. The substrings do not overlap, but they can be separated from each other or from the ends of the attribute value by zero or more string elements. However, at least one attribute of type **DS_ATTRIBUTE_VALUES**, **DS_INITIAL_SUBSTRING**, or **DS_FINAL_SUBSTRING** must exist.

- **DS_FINAL_SUBSTRING**

If present, this attribute is the substring that will match the final part of an attribute value in the entry. This attribute can only exist if the **DS_FILTER_ITEM_TYPE** is equal to **DS_SUBSTRINGS**.

- **DS_INITIAL_SUBSTRING**

If present, this attribute is the substring that will match the initial part of an attribute value in the entry. This attribute can only exist if the **DS_FILTER_ITEM_TYPE** is equal to **DS_SUBSTRINGS**.

DS_C_LIBRARY_ERROR

An instance of OM class **DS_C_LIBRARY_ERROR** reports an error detected by the interface function library.

An application is not permitted to create or modify instances of this OM class. An instance of this OM class has the OM attributes of its superclasses, *OM_C_OBJECT* and *DS_C_ERROR*, and no additional OM attributes.

Each function has several possible errors that can be detected by the library itself and that are returned directly by the subroutine. These errors occur when the library cannot perform an action, submit a service request, or decipher a response from the Directory Service.

The OM attribute **DS_PROBLEM**, which is inherited from the superclass *DS_C_ERROR*, identifies the particular library error that occurred. (In the *z/OS DCE Application Development Reference*, the ERRORS section of each function description lists the errors that the respective function can return.) Its value is one of the following:

- **DS_E_BAD_ARGUMENT**

An invalid argument (other than *name*) was supplied. Use of an instance of OM class **DS_C_ATTRIBUTE** with no values of the OM attribute **DS_ATTRIBUTE_VALUES** as an input argument to a Directory Service function results in this error. Directory attributes always have at least one value.

- **DS_E_BAD_CLASS**

The OM class of an argument is not supported for this operation.

- **DS_E_BAD_CONTEXT**

An invalid *context* parameter was supplied.

- **DS_E_BAD_NAME**

An invalid *name* parameter was supplied.

- **DS_E_BAD_SESSION**

An invalid *session* parameter was supplied.

- **DS_E_MISCELLANEOUS**

A miscellaneous error occurred in interacting with the Directory Service. This error is returned if the interface cannot clear a transient system error by retrying the affected system call.

- **DS_E_MISSING_TYPE**

The attribute type is not included in an AVA that is passed as part of a distinguished name argument.

- **DS_E_MIXED_SYNCHRONOUS**

An attempt is made to start a synchronous operation when there are outstanding asynchronous operations.

- **DS_E_NOT_SUPPORTED**

An attempt is made to use an optional function that is not available in this implementation.

- **DS_E_TOO_MANY_OPERATIONS**

No more Directory Service operations can be performed until at least one asynchronous operation is completed.

- **DS_E_TOO_MANY_SESSIONS**

No more Directory Service sessions can be started.

Note: Both **DS_E_MIXED_SYNCHRONOUS** and **DS_E_TOO_MANY_OPERATIONS** are asynchronous errors. These two errors are not returned by z/OS DCE.

DS_C_LIST_INFO

An instance of OM class **DS_C_LIST_INFO** is part of the results of **ds_list()**.

An application is not permitted to create or modify instances of this OM class. An instance of this OM class has the OM attributes of its superclasses, *OM_C_OBJECT* and *DS_C_COMMON_RESULTS*, in addition to the OM attributes listed in Table 40.

Table 40 (Page 1 of 2). OM Attributes of DS_C_LIST_INFO

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_OBJECT_NAME	Object(<i>DS_C_NAME</i>)	-	0 or 1	-

Table 40 (Page 2 of 2). OM Attributes of DS_C_LIST_INFO

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_PARTIAL_OUTCOME_QUAL	Object(DS_C_PARTIAL_OUTCOME_QUAL)	-	0 or 1	-
DS_SUBORDINATES	Object(DS_C_LIST_INFO_ITEM)	-	0 or more	-

- **DS_OBJECT_NAME**

This attribute is the distinguished name of the target object of the operation. It is present if the OM attribute **DS_ALIAS_DEREFERENCED**, inherited from the superclass *DS_C_COMMON_RESULTS*, is **OM_TRUE**.

- **DS_PARTIAL_OUTCOME_QUAL**

This OM attribute value is present if the list of subordinates is incomplete. The DSA or DSAs that provided this list did not complete the search for some reason. The partial outcome qualifier contains details of why the search was not completed and which areas of the directory have not been searched.

- **DS_SUBORDINATES**

This attribute contains information about zero or more subordinate objects identified by **ds_list()**.

DS_C_LIST_INFO_ITEM

An instance of OM class **DS_C_LIST_INFO_ITEM** comprises details returned by **ds_list()** of a single subordinate object.

An application is not permitted to create or modify instances of this OM class. An instance of this OM class has the OM attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attributes listed in Table 41.

Table 41. OM Attributes of DS_C_LIST_INFO_ITEM

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_ALIAS_ENTRY	OM_S_BOOLEAN	-	1	-
OM_S_BOOLEAN	OM_S_BOOLEAN	-	1	-
DS_RDN	Object(DS_C_RELATIVE_NAME)	-	1	-

- **DS_ALIAS_ENTRY**

This attribute indicates whether the object is an alias.

- **DS_FROM_ENTRY**

This attribute indicates whether information about the object was obtained directly from its directory entry, rather than from a copy of the entry.

- **DS_RDN**

This attribute contains the RDN of the object. If this is the name of an alias entry, as indicated by **DS_ALIAS_ENTRY**, it is not dereferenced.

DS_C_LIST_RESULT

An instance of OM class **DS_C_LIST_RESULT** comprises the results of a successful call to **ds_list()**.

An application is not permitted to create or modify instances of this OM class. An instance of this OM class has the OM attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attributes listed in Table 42.

Table 42. OM Attributes of *DS_C_LIST_RESULT*

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_LIST_INFO	Object(DS_C_LIST_INFO)	-	0 or 1	-
DS_UNCORRELATED_LIST_INFO	Object(DS_C_LIST_RESULT)	-	0 or more	-

Note: No instance contains values of both OM attributes.

- **DS_LIST_INFO**

This attribute contains the full results of **ds_list()**, or just part of them.

- **DS_UNCORRELATED_LIST_INFO**

When the DUA requests a protection request of *signed*, the information returned can consist of a number of sets of results originating from, and signed by, different components of the directory. Implementations can reflect this structure by nesting **DS_LIST_RESULT** OM objects as values of this OM attribute. Alternatively, they can collapse all results into a single value of the OM attribute **DS_LIST_INFO**. The DCE Directory Service does not support the optional feature of signed results; therefore, this OM attribute is never present.

DS_C_NAME

The OM class *DS_C_NAME* represents a name of an object in the directory, or a part of such a name.

It is an abstract class, that has the attributes of its superclass, *OM_C_OBJECT*, and no other OM attributes.

A name uniquely distinguishes the object from all other objects whose entries are displayed in the DIT. However, an object can have more than one name; that is, a name need not be unique. A DN is unique; there are no other DNs that identify the same object. An RDN is part of a name, and only distinguishes the object from others that are its siblings.

Most of the interface functions take a *name* parameter, the value of which must be an instance of one of the subclasses of this OM class. Thus, this OM class is useful for amalgamating all possible representations of names.

The DCE XDS implementation defines one subclass of this OM class, and thus, a single representation for names; that is **DS_C_DS_DN**, which provides a representation for names, including distinguished names.

DS_C_NAME_ERROR

An instance of OM class **DS_C_NAME_ERROR** reports a name-related Directory Service error.

An application is not permitted to create or modify instances of this OM class. An instance of this OM class has the OM attributes of its superclasses, *OM_C_OBJECT* and *DS_C_ERROR*, in addition to the OM attributes listed in Table 43.

Table 43. OM Attributes of *DS_C_NAME_ERROR*

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_MATCHED	Object(<i>DS_C_NAME</i>)	-	1	-

- **DS_MATCHED**

This attribute identifies the initial part (up to, but excluding, the first RDN that is unrecognized) of the name that is supplied, or of the name resulting from dereferencing an alias. It names the lowest entry (object or alias) in the DIT that is matched.

The OM attribute **DS_PROBLEM**, which is inherited from the superclass *DS_C_ERROR*, identifies the cause of the failure. Its value is one of the following:

- **DS_E_ALIAS_DEREFERENCING_PROBLEM**

An alias is encountered where an alias is not permitted, for example, in a modification operation when the **DS_DONT_DEREFERENCE_ALIASES** service control is set or when one alias points to another alias.

- **DS_E_ALIAS_PROBLEM**

An alias is dereferenced that names an object that does not exist, that is, for which no directory entry can be found.

- **DS_E_INVALID_ATTRIBUTE_VALUE**

The attribute value in an AVA of an RDN contained in the name does not conform to the attribute syntax prescribed for the attribute type in the AVA. This problem is called **invalidAttributeSyntax** in the standards, but that name is used only for a **DS_C_ATTRIBUTE_PROBLEM** in this interface.

- **DS_E_NO_SUCH_OBJECT**

The specified name does not match the name of any object in the directory.

DS_C_OPERATION_PROGRESS

An instance of OM class **DS_C_OPERATION_PROGRESS** specifies the progress or processing state of a directory request.

An application is not permitted to create or modify instances of this OM class. An instance of this OM class has the OM attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attributes listed in Table 44.

Table 44. OM Attributes of DS_C_OPERATION_PROGRESS

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_NAME_RESOLUTION_PHASE	Enum(DS_Name_Resolution_Phase)	-	1	-
DS_NEXT_RDN_TO_BE_RESOLVED	OM_S_INTEGER	-	0 or 1	-

The target name is the name upon which processing of the directory request is currently focused.

- **DS_NAME_RESOLUTION_PHASE**

This attribute indicates what phase is reached in handling the target name. It must have one of the following values:

- **DS_COMPLETED** indicates that the DSA holding the target object is reached.
- **DS_NOT_STARTED** indicates that so far a DSA is not reached with a naming context containing the initial RDNs of the name.
- **DS_PROCEEDING** indicates that the initial part of the name has been recognized, although the DSA holding the target object has not yet been reached.

- **DS_NEXT_RDN_TO_BE_RESOLVED**

This attribute indicates to the DSA which of the RDNs in the target name is next to be resolved. It takes the form of an integer in the range from 1 to the number of RDNs in the name. This OM attribute only has a value if the value of **DS_NAME_RESOLUTION_PHASE** is **DS_PROCEEDING**.

The constant **DS_OPERATION_NOT_STARTED** can be used in the **DS_C_CONTEXT** of an operation instead of an instance of this OM class.

DS_C_PARTIAL_OUTCOME_QUAL

An instance of OM class **DS_C_PARTIAL_OUTCOME_QUAL** explains to what extent the results of a call to **ds_list()** or **ds_search()** are incomplete and the reason.

An application is not permitted to create or modify instances of this OM class. An instance of this OM class has the OM attributes of its superclass, **OM_C_OBJECT**, in addition to the OM attributes listed in Table 45.

Table 45. OM Attributes of DS_C_PARTIAL_OUTCOME_QUAL

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_LIMIT_PROBLEM	Enum(DS_Limit_Problem)	-	1	-
DS_UNAVAILABLE_CRIT_EXT	OM_S_BOOLEAN	-	1	-
DS_UNEXPLORED	Object(DS_C_CONTINUATION_REF)	-	0 or more	-

- **DS_LIMIT_PROBLEM**

This attribute explains fully or partly why the results are incomplete. It can have one of the following values:

- **DS_ADMINISTRATIVE_LIMIT_EXCEEDED** indicates that an administrative limit is reached.
- **DS_NO_LIMIT_EXCEEDED** indicates that there is no limit problem.

- **DS_SIZE_LIMIT_EXCEEDED** indicates that the maximum number of objects specified as a service control is reached.
- **DS_TIME_LIMIT_EXCEEDED** indicates that the maximum number of seconds specified as a service control is reached.

- **DS_UNAVAILABLE_CRIT_EXT**

If **OM_TRUE**, this attribute indicates that some part of the Directory Service cannot provide a requested critical service extension. The user requested one or more standard service extensions by including values of the OM attribute **DS_EXT** in the **DS_C_CONTEXT** supplied for the operation. Furthermore, the user indicated that some of these extensions are essential by setting the OM attribute **DS_CRIT** in the extension to **OM_TRUE**. Some of the critical extensions cannot be performed by one particular DSA or by a number of DSAs. In general, it is not possible to determine which DSA could not perform which particular extension.

Note: Because **DS_EXT** is not supported, this OM attribute will always be set to **OM_FALSE**.

- **DS_UNEXPLORED**

This attribute identifies any regions of the Directory that are left unexplored in a way that the directory request can be continued. Only continuation references within the scope specified by the **DS_SCOPE_OF_REFERRAL** service control are included.

DS_C_PRESENTATION_ADDRESS

An instance of OM class **DS_C_PRESENTATION_ADDRESS** is a presentation address of an OSI application entity, which is used for OSI communications with this instance.

An instance of this OM class has the OM attributes of its superclasses, *OM_C_OBJECT* and *DS_C_ADDRESS*, in addition to the OM attributes listed in Table 46.

Table 46. OM Attributes of *DS_C_PRESENTATION_ADDRESS*

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_N_ADDRESSES	String(OM_S_OCTET_STRING)	-	1 or more	-
DS_P_SELECTOR	String(OM_S_OCTET_STRING)	-	0 or 1	-
DS_S_SELECTOR	String(OM_S_OCTET_STRING)	-	0 or 1	-
DS_T_SELECTOR	String(OM_S_OCTET_STRING)	-	0 or 1	-

- **DS_N_ADDRESSES**

This attribute is the network addresses of the application entity.

- **DS_P_SELECTOR**

This attribute is the presentation selector.

- **DS_S_SELECTOR**

This attribute is the session selector.

- **DS_T_SELECTOR**

This attribute is the transport selector.

DS_C_READ_RESULT

An instance of OM class **DS_C_READ_RESULT** comprises the result of a successful call to **ds_read()**.

An application is not permitted to create or modify instances of this OM class. An instance of this OM class has the OM attributes of its superclasses, *OM_C_OBJECT* and *DS_C_COMMON_RESULTS*, in addition to the OM attributes listed in Table 47.

Table 47. OM Attributes of *DS_C_READ_RESULT*

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_ENTRY	Object(DS_C_ENTRY_INFO)	-	1	-

- **DS_ENTRY**

This attribute contains the information extracted from the directory entry of the target object.

DS_C_REFERRAL

An instance of OM class **DS_C_REFERRAL** reports failure to perform an operation and redirects the requestor to one or more access points better equipped to perform the operation.

An application is not permitted to create or modify instances of this OM class. An instance of this OM class has the OM attributes of its superclasses, *OM_C_OBJECT* and *DS_C_CONTINUATION_REF*, and no additional OM attributes.

The referral is a continuation reference by means of which the operation can proceed.

DS_C_RELATIVE_NAME

The OM class *DS_C_RELATIVE_NAME* represents the RDNs of objects in the Directory. It is an abstract class that has the attributes of its superclass, *OM_C_OBJECT*, and no other OM attributes.

An RDN is part of a name, and only distinguishes the object from others that are its siblings. This OM class is used to accumulate all possible representations of RDNs. An argument of interface functions that is an RDN, or an OM attribute value that is an RDN, is an instance of one of the subclasses of this OM class.

The DCE XDS API defines one subclass of this OM class, and thus, a single representation for RDNs, that is, **DS_C_DS_RDN**, which provides a representation for RDNs.

DS_C_SEARCH_INFO

Note: Because **DS_C_SEARCH_INFO** is GDS-specific, it is not supported.

An instance of OM class **DS_C_SEARCH_INFO** is part of the results of **ds_search()**.

An application is not permitted to create or modify instances of this OM class. An instance of this OM class has the OM attributes of its superclasses, *OM_C_OBJECT* and *DS_C_COMMON_RESULTS*, in addition to the OM attributes listed in Table 48.

Table 48. OM Attributes of DS_C_SEARCH_INFO

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_ENTRIES	Object(DS_C_ENTRY_INFO)	-	0 or more	-
DS_OBJECT_NAME	Object(DS_C_NAME)	-	0 or 1	-
DS_PARTIAL_OUTCOME_QUAL	Object(DS_C_PARTIAL_OUTCOME_QUAL)	-	0 or 1	-

- **DS_ENTRIES**

This attribute contains information about zero or more objects found by **ds_search()** that matched the given selection criteria.

- **DS_OBJECT_NAME**

This attribute contains the distinguished name of the target object of the operation. It is present if the OM attribute **DS_ALIAS_DEREFERENCED**, inherited from the superclass, *DS_C_COMMON_RESULTS*, is **OM_TRUE**.

- **DS_PARTIAL_OUTCOME_QUAL**

This OM attribute value is only present if the list of entries is incomplete. The DSA or DSAs that provided this list did not complete the search for some reason. The partial outcome qualifier contains details of why the search was not completed and which areas of the directory were not searched.

DS_C_SEARCH_RESULT

Note: Because **DS_C_SEARCH_RESULT** is GDS-specific, it is not supported.

An instance of OM class **DS_C_SEARCH_RESULT** comprises the results of a successful call to **ds_search()**.

An application is not permitted to create or modify instances of this OM class. An instance of this OM class has the OM attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attributes listed in Table 49.

Table 49. OM Attributes of DS_C_SEARCH_RESULT

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_SEARCH_INFO	Object(DS_C_SEARCH_INFO)	-	0 or 1	-
DS_UNCORRELATED_SEARCH_INFO	Object(DS_C_SEARCH_RESULT)	-	0 or more	-

Note: No instance contains values of both OM attributes.

- **DS_SEARCH_INFO**

This attribute contains the full results of **ds_search()**, or part of the results.

- **DS_UNCORRELATED_SEARCH_INFO**

When the DUA requests a protection request of *signed*, the information returned can consist of a number of sets of results originating from and signed by different components of the Directory Service. Implementations can reflect this structure by nesting **DS_C_SEARCH_RESULT** OM objects as values

of this OM attribute. Alternatively, they can collapse all results into a single value of the OM attribute **DS_SEARCH_INFO**.

DS_C_SECURITY_ERROR

An instance of OM class **DS_C_SECURITY_ERROR** reports a security-related Directory Service error.

An application is not permitted to create or modify instances of this OM class. An instance of this OM class has the OM attributes of its superclasses, *OM_C_OBJECT* and *DS_C_ERROR*, and no additional OM attributes.

The OM attribute **DS_PROBLEM**, which is inherited from the superclass *DS_C_ERROR*, identifies the cause of this failure. Its value is one of the following:

- **DS_E_INAPPROP_AUTHENTICATION**

The level of security attached to the requestor's credentials is inconsistent with the level of protection requested; for example, simple credentials are supplied whereas strong credentials are required.

- **DS_E_INSUFFICIENT_ACCESS_RIGHTS**

The requestor does not have permission to perform the operation. A **ds_read()** operation only returns this error when access rights preclude the reading of all requested attribute values.

- **DS_E_INVALID_CREDENTIALS**

The requestor's credentials are invalid.

- **DS_E_INVALID_SIGNATURE**

The signature affixed to the request is invalid.

- **DS_E_NO_INFO**

The request produced a security error for which no other information is available.

- **DS_E_PROTECTION_REQUIRED**

The Directory Service cannot perform the operation because it is unsigned.

DS_C_SERVICE_ERROR

An instance of OM class **DS_C_SERVICE_ERROR** reports a Directory Service error related to the provision of the service.

An application is not permitted to create or modify instances of this OM class. An instance of this OM class has the OM attributes of its superclasses, *OM_C_OBJECT* and *DS_C_ERROR*, and no additional OM attributes.

The OM attribute **DS_PROBLEM**, which is inherited from the superclass *DS_C_ERROR*, identifies the cause of the failure. Its value is one of the following:

- **DS_E_ADMIN_LIMIT_EXCEEDED**

The operation could not be performed within the administrative constraints on the directory, and no partial results are available.

- **DS_E_BUSY**

Some part of the Directory Service is temporarily too busy to perform the operation, but will be available after a short while.

- **DS_E_CHAINING_REQUIRED**

Chaining is required to perform the operation, but is prohibited by the **DS_CHAINING_PROHIBITED** service control.

- **DS_E_DIT_ERROR**

An inconsistency is detected in the DIT that can be localized to a particular entry or set of entries.

- **DS_E_INVALID_REF**

The DSA cannot perform the request as directed, that is through **DS_C_OPERATION_PROGRESS** in the **DS_C_CONTEXT**. The cause can be an invalid referral.

- **DS_E_LOOP_DETECTED**

A DSA detected a loop within the directory.

- **DS_E_OUT_OF_SCOPE**

The Directory Service cannot provide a referral or partial outcome qualifier within the required scope.

- **DS_E_TIME_LIMIT_EXCEEDED**

The operation could not be performed within the time specified by the **DS_TIME_LIMIT** service control, and no partial results are available.

- **DS_E_UNABLE_TO_PROCEED**

A DSA without administrative authority over a particular naming context is asked to resolve a name in that context.

- **DS_E_UNAVAILABLE**

Some part of the directory is not currently available.

- **DS_E_UNAVAILABLE_CRIT_EXT**

One or more critical extensions are requested, but are not available.

- **DS_E_UNWILLING_TO_PERFORM**

Some part of the Directory Service cannot perform the operation because it requires excessive resources, or because doing so violates administrative policy.

DS_C_SESSION

An instance of OM class **DS_C_SESSION** identifies a particular link from the application program to a DUA.

An instance of this OM class has the OM attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attributes listed in Table 50

Table 50 (Page 1 of 2). OM Attributes of DS_C_SESSION

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_DSA_ADDRESS	Object(<i>DS_C_ADDRESS</i>)	-	0 or 1	<i>local</i> ¹
DS_DSA_NAME	Object(<i>DS_C_NAME</i>)	-	0 or 1	<i>local</i> ¹
DS_FILE_DESCRIPTOR	OM_S_INTEGER	-	1	see the text

Table 50 (Page 2 of 2). OM Attributes of DS_C_SESSION

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_REQUESTOR	Object(DS_C_NAME)	-	0 or 1	-

Notes:

1. The default values of these OM attributes are administered locally.

The **DS_C_SESSION** gathers all the information that describes a particular directory interaction. The parameters that control such a session are set up in an instance of this OM class, which is then passed as an argument to **ds_bind()**. This sets the OM attributes that describe the actual characteristics of this session, and then starts the session. A session started in this way must pass as the first argument to each interface function. The result of modifying an initiated session is unspecified. Finally, **ds_unbind()** is used to terminate the session, after which the parameters can be modified and a new session started using the same instance, if required. Multiple concurrent sessions can run using multiple instances of this OM class.

The OM attributes of a session are as follows:

- **DS_DSA_ADDRESS**
This attribute indicates the address of the default DSA named by **DS_DSA_NAME**.
- **DS_DSA_NAME**
This attribute indicates the distinguished name of the DSA that is used by default to service directory requests.
- **DS_FILE_DESCRIPTOR** (NOT SUPPORTED)
This OM attribute is not used by DCE XDS and is always set to **DS_NO_VALID_FILE_DESCRIPTOR**.
- **DS_REQUESTOR**
This attribute is the distinguished name of the user of this Directory Service session.

Applications can assume that an object of OM class **DS_C_SESSION**, created with default values of all its OM attributes, works with all the interface functions. Local administrators need to ensure that this assumption is true. Such a session can be created by passing the constant **DS_DEFAULT_SESSION** as an argument to **ds_bind()**.

DS_C_SYSTEM_ERROR

An instance of OM class **DS_C_SYSTEM_ERROR** reports an error that occurred in the underlying operating system.

An application is not permitted to create or modify instances of this OM class. An instance of this OM class has the OM attributes of its superclasses, **OM_C_OBJECT** and **DS_C_ERROR**, and no additional OM attributes, although there can be additional implementation-defined OM attributes.

The OM attribute **DS_PROBLEM**, which is inherited from the superclass **DS_C_ERROR**, identifies the cause of the failure. Its value is the same as that of **errno** defined in the C language.

The standard names of system errors are defined in Volume 2 of the *X/Open Portability Guide*.

If such an error persists, a **DS_C_LIBRARY_ERROR** (**DS_E_MISCELLANEOUS**) is reported.

DS_C_UPDATE_ERROR

An instance of OM class **DS_C_UPDATE_ERROR** reports a Directory Service error peculiar to a modification operation.

An application is not permitted to create or modify instances of this OM class. An instance of this OM class has the OM attributes of its superclasses, *OM_C_OBJECT* and *DS_C_ERROR*, and no additional OM attributes.

The OM attribute **DS_PROBLEM**, which is inherited from the superclass *DS_C_ERROR*, identifies the cause of the failure. Its value is one of the following:

- **DS_E_AFFECTS_MULTIPLE_DSAS**

The modification affects several DSAs, and such a modification is prohibited. Local agreement between DSAs can allow modifications that affect multiple DSAs, for example, adding entries whose immediate superior entry is in a different DSA. This problem is not reported in such cases.

- **DS_E_ENTRY_EXISTS**

The name passed to **ds_add_entry()** already exists.

- **DS_E_NAMING_VIOLATION**

The modification leaves the DIT structured incorrectly. That is, it adds an entry as the subordinate of an alias, or in a region of the DIT not permitted to a member of its object class, or it defines an RDN that includes a forbidden attribute type.

- **DS_E_NOT_ALLOWED_ON_NON_LEAF**

The modification would be to an interior node of the DIT, and such a modification is prohibited.

- **DS_E_NOT_ALLOWED_ON_RDN**

The modification alters an object's RDN.

- **DS_E_OBJECT_CLASS_MOD_PROHIB**

The modification alters an entry's object class attribute.

- **DS_E_OBJECT_CLASS_VIOLATION**

The modification leaves a directory entry inconsistent with its object class definition.

Chapter 12. Basic Directory Contents Package

The standards define a number of attribute types (known as the *selected attribute types*), attribute syntaxes, attribute sets, and object classes (known as the *selected object classes*). These definitions allow the creation and maintenance of directory entries for a number of common objects so that the representation of all such objects is the same throughout the directory.¹ They include such objects as Country, Person, and Organization.

Note: z/OS DCE does not support GDS. The GDS naming information presented is intended to increase your understanding of DCE name structure and concepts.

This chapter outlines names for each of these items, and defines OM classes to represent those that are not represented directly by OM syntaxes. The values of attributes in the directory are not restricted to those discussed in this chapter, and new attribute types and syntaxes can be created at any time. (For further information on how the values of other syntaxes are represented in the interface, see “Attribute and Attribute Value Assertion” on page 237.)

The constants and OM classes in this chapter are defined in addition to those in Chapter 11, “XDS Class Definitions” on page 241, because they are not essential to the working of the interface, but instead allow directory entries to be utilized. The definitions belong to the Basic Directory Contents Package (BDCP), which is supported by the DCE XDS API following negotiation of its use with **ds_version()**.

The object identifier associated with the BDC Package is **{iso(1) identified-organization(3) icd-ecma(0012) member-company(2) dec(1011) xopen(28) bdc(1)}** with the following encoding:

```
\x2B\x0C\x02\x87\x73\x1C\x01
```

This identifier is represented by the constant **DS_BASIC_DIRECTORY_CONTENTS_PKG**. The C constants associated with this package are in the **xdsbdc.h** header file. (See the *z/OS DCE Application Development Reference* to see the contents of this header file.)

The concepts and notation used are introduced in the introductory section of Chapter 11, “XDS Class Definitions” on page 241. A complete explanation of the meaning of the attributes and object classes is not given. The purpose here is simply to present the representation of these items in the interface.

The selected attribute types are presented first, followed by the selected object classes. Next, the OM class hierarchy and OM class definitions required to support the selected attribute types are presented.

Selected Attribute Types

This section presents the attribute types defined in the standards to be used in directory entries. Each directory entry is composed of a number of attributes, each of which comprises an attribute type together with one or more attribute values. The form of each value of an attribute is determined by the attribute syntax associated with the attribute’s type.

In the interface, attributes are displayed as instances of OM class **DS_C_ATTRIBUTE** with the attribute type represented as the value of the OM attribute **DS_ATTRIBUTE_TYPE**, and the attribute value (or values) represented as the value (or values) of the OM attribute **DS_ATTRIBUTE_VALUES**. Each attribute type has an object identifier, assigned in the standards, which is the value of the OM attribute

¹ These definitions are chiefly in *The Directory: Selected Attribute Types* {ISO 9594-6, CCITT X.520} and *The Directory: Selected Object Classes* {ISO 9594-7, CCITT X.521}, with additional material in *The Directory: Overview of Concepts, Models and Services* {ISO 9594-1, CCITT X.500} and *The Directory: Authentication Framework* {ISO 9594-8, CCITT X.509}.

DS_ATTRIBUTE_TYPE. These object identifiers are represented in the interface by constants with the same name as the directory attribute, and are prefixed with **DS_A_** so that they can be easily identified.

Table 51 shows the names of the attribute types defined in the standards, together with the Basic Encoding Rules (BERs) for encoding the object identifiers associated with each of them. Table 52 on page 277 shows the names of the attribute types, together with the OM Value Syntax that is used in the interface to represent values of that attribute type. Table 52 on page 277 also includes the range of lengths permitted for the string types. This indicates whether the attribute can have multiple values and which matching rules are provided for the syntax. Following the table is a brief description of each attribute.

Note: Referring to Table 52 on page 277, any attribute of an entry in the CDS namespace that has been added using the X/Open interface is assumed to be multi-valued, and the length and matching rules do not apply (except for **DS_A_ALIASED_OBJECT_NAME**). Some of these attributes cannot be added to CDS as noted in Table 52.

The standards define matching rules that are used for deciding whether two values are equal (E), for ordering (O) two values, and for identifying one value as a substring (S) of another in Directory Service operations. Specific matching rules are given in this chapter for certain attributes. In addition, the following general rules apply as indicated:

- Differences between attribute values whose syntax is String(**OM_S_NUMERIC_STRING**), String(**OM_S_PRINTABLE_STRING**), or String(**OM_S_TELETEX_STRING**) are considered insignificant for the following reasons:
 - Differences caused by the presence of spaces preceding the first printing character
 - Spaces following the last printing character
 - More than one consecutive space anywhere within the value.
- For all attribute values whose syntax is String(**OM_S_TELETEX_STRING**), differences in the case of alphabetical characters are considered insignificant.

Note: The third and fourth columns of Table 51 contain the contents octets of the BER encoding of the object identifier. All these object identifiers stem from the root **{joint-iso-ccitt(2) ds(5) attributeType(4)}**. Basic encoding rules state that the first two decimal numbers be combined according to the formula $x*40+y$ to form the first hexadecimal value. Thus $2.5.4 \approx 85.4 \approx \backslashx55\backslashx04$.

Table 51 (Page 1 of 2). Object Identifiers for Selected Attribute Types

Package	Attribute Type	Object Identifier BER	
		DECIMAL	HEXADECIMAL
BDCP	DS_A_ALIASED_OBJECT_NAME	85.4.1	\x55\x04\x01
BDCP	DS_A_BUSINESS_CATEGORY	85.4.15	\x55\x04\x0F
BDCP	DS_A_COMMON_NAME	85.4.3	\x55\x04\x03
BDCP	DS_A_COUNTRY_NAME	85.4.6	\x55\x04\x06
BDCP	DS_A_DESCRIPTION	85.4.13	\x55\x04\x0D
BDCP	DS_A_DEST_INDICATOR	85.4.27	\x55\x04\x1B
BDCP	DS_A_FACSIMILE_PHONE_NBR	85.4.23	\x55\x04\x17
BDCP	DS_A_INTERNAT_ISDN_NBR	85.4.25	\x55\x04\x19
BDCP	DS_A_KNOWLEDGE_INFO	85.4.2	\x55\x04\x02
BDCP	DS_A_LOCALITY_NAME	85.4.7	\x55\x04\x07

Table 51 (Page 2 of 2). Object Identifiers for Selected Attribute Types

Package	Attribute Type	Object Identifier BER	
		DECIMAL	HEXADECIMAL
BDCP	DS_A_MEMBER	85.4.31	\x55\x04\x1F
BDCP	DS_A_OBJECT_CLASS	85.4.0	\x55\x04\x00
BDCP	DS_A_ORG_NAME	85.4.10	\x55\x04\x0A
BDCP	DS_A_ORG_UNIT_NAME	85.4.11	\x55\x04\x0B
BDCP	DS_A_OWNER	85.4.32	\x55\x04\x20
BDCP	DS_A_PHYS_DELIV_OFF_NAME	85.4.19	\x55\x04\x13
BDCP	DS_A_POST_OFFICE_BOX	85.4.18	\x55\x04\x12
BDCP	DS_A_POSTAL_ADDRESS	85.4.16	\x55\x04\x10
BDCP	DS_A_POSTAL_CODE	85.4.17	\x55\x04\x11
BDCP	DS_A_PREF_DELIV_METHOD	85.4.28	\x55\x04\x1C
BDCP	DS_A_PRESENTATION_ADDRESS	85.4.29	\x55\x04\x1D
BDCP	DS_A_REGISTERED_ADDRESS	85.4.26	\x55\x04\x1A
BDCP	DS_A_ROLE_OCCUPANT	85.4.33	\x55\x04\x21
BDCP	DS_A_SEARCH_GUIDE	85.4.14	\x55\x04\x0E
BDCP	DS_A_SEE_ALSO	85.4.34	\x55\x04\x22
BDCP	DS_A_SERIAL_NBR	85.4.5	\x55\x04\x05
BDCP	DS_A_STATE_OR_PROV_NAME	85.4.8	\x55\x04\x08
BDCP	DS_A_STREET_ADDRESS	85.4.9	\x55\x04\x09
BDCP	DS_A_SUPPORT_APPLIC_CONTEXT	85.4.30	\x55\x04\x1E
BDCP	DS_A_SURNAME	85.4.4	\x55\x04\x04
BDCP	DS_A_PHONE_NBR	85.4.20	\x55\x04\x14
BDCP	DS_A_TELETEX_TERM_IDENT	85.4.22	\x55\x04\x16
BDCP	DS_A_TELEX_NBR	85.4.21	\x55\x04\x15
BDCP	DS_A_TITLE	85.4.12	\x55\x04\x0C
BDCP	DS_A_USER_PASSWORD	85.4.35	\x55\x04\x23
BDCP	DS_A_X121_ADDRESS	85.4.24	\x55\x04\x18

Table 52 (Page 1 of 3). Representation of Values for Selected Attribute Types

Attribute Type	OM Value Syntax	Value Length	Multi-Valued	Matching Rules
DS_A_ALIASED_OBJECT_NAME	Object(DS_C_NAME)	-	No	E
DS_A_BUSINESS_CATEGORY	String(OM_S_TELETEX_STRING)	1-128	Yes	E, S
DS_A_COMMON_NAME	String(OM_S_TELETEX_STRING)	1-64	Yes	E, S
DS_A_COUNTRY_NAME	String(OM_S_PRINTABLE_STRING) ¹	2	No	E
DS_A_DESCRIPTION	String(OM_S_TELETEX_STRING)	1-1024	Yes	E, S
DS_A_DEST_INDICATOR	String(OM_S_PRINTABLE_STRING) ²	1-128	Yes	E, S

Table 52 (Page 2 of 3). Representation of Values for Selected Attribute Types

Attribute Type	OM Value Syntax	Value Length	Multi-Valued	Matching Rules
DS_A_FACSIMILE_PHONE_NBR	Object(DS_C_FACSIMILE_PHONE_NBR)	-	Yes	
DS_A_INTERNAT_ISDN_NBR	String(OM_S_NUMERIC_STRING) ³	1-16	Yes	-
DS_A_KNOWLEDGE_INFO	String(OM_S_TELETEX_STRING)	-	Yes	E, S
DS_A_LOCALITY_NAME	String(OM_S_TELETEX_STRING)	1-128	Yes	E, S
DS_A_MEMBER	Object(DS_C_NAME)	-	Yes	E
DS_A_OBJECT_CLASS	String(OM_S_OBJECT_IDENTIFIER_STRING)	-	yes	E
DS_A_ORG_NAME	String(OM_S_TELETEX_STRING)	1-64	Yes	E, S
DS_A_ORG_UNIT_NAME	String(OM_S_TELETEX_STRING)	1-64	Yes	E, S
DS_A_OWNER	Object(DS_C_NAME)	-	Yes	E
DS_A_PHYS_DELIV_OFF_NAME	String(OM_S_TELETEX_STRING)	1-128	Yes	E, S
DS_A_POST_OFFICE_BOX	String(OM_S_TELETEX_STRING)	1-40	Yes	E, S
DS_A_POSTAL_ADDRESS	Object(DS_C_POSTAL_ADDRESS)	-	Yes	E
DS_A_POSTAL_CODE	String(OM_S_TELETEX_STRING)	1-40	Yes	E, S
DS_A_PREF_DELIV_METHOD	Enum(DS_Preferred_Delivery_Method)	-	Yes	-
DS_A_PRESENTATION_ADDRESS	Object(DS_C_PRESENTATION_ADDRESS)	-	No	E
DS_A_REGISTERED_ADDRESS	Object(DS_C_POSTAL_ADDRESS)	-	Yes	
DS_A_ROLE_OCCUPANT	Object(DS_C_NAME)	-	Yes	E
DS_A_SEARCH_GUIDE	Object(DS_C_SEARCH_GUIDE)	-	Yes	
DS_A_SEE_ALSO	Object(DS_C_NAME)	-	Yes	E
DS_A_SERIAL_NBR	String(OM_S_PRINTABLE_STRING)	1-64	Yes	E, S
DS_A_STATE_OR_PROV_NAME	String(OM_S_TELETEX_STRING)	1-128	Yes	E, S
DS_A_STREET_ADDRESS	String(OM_S_TELETEX_STRING)	1-128	Yes	E, S
DS_A_SUPPORT_APPLIC_CONTEXT	String(OM_S_OBJECT_IDENTIFIER_STRING)	-	Yes	E
DS_A_SURNAME	String(OM_S_TELETEX_STRING)	1-64	Yes	E, S
DS_A_PHONE_NBR	String(OM_S_PRINTABLE_STRING) ⁴	1-32	Yes	E, S
DS_A_TELETEX_TERM_IDENT	Object(DS_C_TELETEX_TERM_IDENT)	-	Yes	
DS_A_TELEX_NBR	Object(DS_C_TELEX_NBR)	-	Yes	
DS_A_TITLE	String(OM_S_TELETEX_STRING)	1-64	Yes	E, S
DS_A_USER_PASSWORD	String(OM_S_OCTET_STRING)	0-128	Yes	-

Table 52 (Page 3 of 3). Representation of Values for Selected Attribute Types

Attribute Type	OM Value Syntax	Value Length	Multi-Valued	Matching Rules
DS_A_X121_ADDRESS	String(OM_S_NUMERIC_STRING) ⁵	1-15	Yes	E, S

Notes:

1. As permitted by ISO 3166.
2. As permitted by CCIT Recommendations F.1 and F.31.
3. As permitted by CCIT E.164.
4. As permitted by CCIT E.123 (for example, +44 582 10101).
5. As permitted by CCIT X.121.

Throughout the descriptions that follow, the term *object* indicates the directory object whose directory entry contains the corresponding directory attributes.

- **DS_A_ALIASED_OBJECT_NAME**

This attribute occurs only in alias entries. It assigns the Distinguished Name (DN) of the object provided with an alias using the entry in which this attribute occurs. An alias is an alternative to an object's DN. Any object can (but need not) have one or more aliases. The Directory Service is said to dereference an alias whenever it replaces the alias during name processing with the distinguished name associated with it by means of this attribute.

- **DS_A_BUSINESS_CATEGORY**

This attribute provides descriptions of the businesses in which the object is engaged.

- **DS_A_COMMON_NAME**

This attribute provides the names by which the object is commonly known in the context defined by its position in the DIT. The names can conform to the naming convention of the country or culture with which the object is associated. They can be ambiguous.

- **DS_A_COUNTRY_NAME**

This attribute identifies the country in which the object is located or with which it is associated in some other important way. The matching rules require that differences in the case of alphabetical characters be considered insignificant. It has a length of two characters, and its values are those listed in ISO 3166.

- **DS_A_DESCRIPTION**

This attribute gives informative descriptions of the object.

- **DS_A_DEST_INDICATOR**

This attribute gives the country-city pairs by which the object can be reached via the public telegram service. The matching rules require that differences in the case of alphabetical characters be considered insignificant.

- **DS_A_FACSIMILE_PHONE_NBR**

This attribute provides the telephone numbers for facsimile terminals (and their parameters, if required) by which the object can be reached or with which it is associated in some other important way.

- **DS_A_INTERNAT_ISDN_NBR**

This attribute provides the international ISDN numbers by means of which the object can be reached or with which it is associated in some other important way. The matching rules require that differences caused by the presence of spaces be considered insignificant.

- **DS_A_KNOWLEDGE_INFO**

This attribute occurs only in entries that describe a DSA. It provides a human-intelligible accumulated description of the directory knowledge possessed by the DSA.

- **DS_A_LOCALITY_NAME**

This attribute identifies geographical areas or localities. When used as part of a directory name, it specifies the localities in which the object is located or with which it is associated in some other important way.

- **DS_A_MEMBER**

This attribute gives the names of objects that are considered members of the present object, for example, a distribution list for electronic mail.

- **DS_A_OBJECT_CLASS**

This attribute identifies the object classes to which the object belongs, and also identifies their superclasses. All such object classes that have object identifiers assigned to them are present, except that object class **DS_O_TOP** need not (but can) be present provided that some other value is present. This attribute must be present in every entry and cannot be modified. For further discussion, see “Selected Object Classes” on page 282.

- **DS_A_ORG_NAME**

This attribute identifies organizations. When used as part of a directory name, it specifies an organization with which the object is affiliated. Several values can identify the same organization in different ways.

- **DS_A_ORG_UNIT_NAME**

This attribute identifies organizational units. When used as part of a directory name, it specifies an organizational unit with which the object is affiliated. The units are understood to be parts of the organization that the **DS_A_ORG_NAME** attribute indicates. Several values can identify the same unit in different ways.

- **DS_A_OWNER**

This attribute gives the names of objects that have responsibility for the object.

- **DS_A_PHYS_DELIV_OFF_NAME**

This attribute gives the names of cities, towns, villages, and so on that contain physical delivery offices through which the object can take delivery of physical mail.

- **DS_A_POST_OFFICE_BOX**

This attribute identifies post office boxes at which the object can take delivery of physical mail. This information is also displayed as part of the **DS_A_POSTAL_ADDRESS** attribute, if it is present.

- **DS_A_POSTAL_ADDRESS**

This attribute gives the postal addresses at which the object can take delivery of physical mail. The matching rules require that differences in the case of alphabetical characters be considered insignificant.

- **DS_A_POSTAL_CODE**

This attribute gives the postal codes assigned to areas or buildings through which the object can take delivery of physical mail. This information is also displayed as part of the **DS_A_POSTAL_ADDRESS** attribute, if it is present.

- **DS_A_PREF_DELIV_METHOD**

This attribute gives the object's preferred methods of communication, in the order of preference. The values are as follows:

- **DS_ANY_DELIV_METHOD** indicates that the object has no preference.
- **DS_G3_FACSIMILE_DELIV** indicates communication using Group 3 facsimile.
- **DS_G4_FACSIMILE_DELIV** indicates delivery using Group 4 facsimile.
- **DS_IA5_TERMINAL_DELIV** indicates delivery using IA5 text.
- **DS_MHS_DELIV** indicates delivery using X.400.
- **DS_PHYS_DELIV** indicates delivery using the postal or other physical delivery system.
- **DS_PHONE_DELIV** indicates delivery using telephone.
- **DS_TELETEX_DELIV** indicates delivery using teletex.
- **DS_TELEX_DELIV** indicates delivery using telex.
- **DS_VIDEOTEX_DELIV** indicates delivery using videotex.

- **DS_A_PRESENTATION_ADDRESS**

This attribute contains the OSI presentation address of the object, which is an OSI application entity. The matching rule for a presented value to match a value stored in the directory is that the P-Selector, S-Selector, and T-Selector of the two presentation addresses must be equal, and the N-Addresses of the presented value must be a subset of those of the stored value.

- **DS_A_REGISTERED_ADDRESS**

This attribute contains mnemonics by which the object can be reached via the public telegram service, according to CCIT Recommendation F.1. A mnemonic identifies an object in the context of a particular city, and is registered in the country containing the city. The matching rules require that differences in the case of alphabetical characters be considered insignificant.

- **DS_A_ROLE_OCCUPANT**

This attribute occurs only in entries that describe an organizational role. It gives the names of objects that fulfill the organizational role.

- **DS_A_SEARCH_GUIDE**

This attribute contains the criteria that can be used to build filters for conducting searches in which the object is the base object.

- **DS_A_SEE_ALSO**

This attribute contains the names of objects that represent other aspects of the real-world object that the present object represents.

- **DS_A_SERIAL_NBR**

This attribute contains the serial numbers of a device.

- **DS_A_STATE_OR_PROV_NAME**

This attribute specifies a state or province. When used as part of a directory name, it identifies states, provinces, or other geographical regions in which the object is located or with which it is associated in some other important way.

- **DS_A_STREET_ADDRESS**

This attribute identifies a site for the local distribution and physical delivery of mail. When used as part of a directory name, it identifies the street address (for example, street name and house number) at which the object is located or with which it is associated in some other important way.

- **DS_A_SUPPORT_APPLIC_CONTEXT**

This attribute occurs only in entries that describe an OSI application entity. It identifies OSI application contexts supported by the object.

- **DS_A_SURNAME**

This attribute occurs only in entries that describe individuals. The surname by which the individual is commonly known, normally inherited from the individual's parent (or parents) or taken at marriage, as determined by the custom of the country or culture with which the individual is associated.

- **DS_A_PHONE_NBR**

This attribute identifies telephones by means of which the object can be reached or with which it is associated in some other important way. The matching rules require that differences caused by the presence of spaces and dashes be considered insignificant.

- **DS_A_TELETEX_TERM_IDENT**

This attribute contains descriptions of teletex terminals by which the object can be reached or with which it is associated in some other important way.

- **DS_A_TELEX_NBR**

This attribute contains descriptions of telex terminals by means of which the object can be reached or with which it is associated in some other important way.

- **DS_A_TITLE**

This attribute identifies positions or functions of the object within its organization.

- **DS_A_USER_PASSWORD**

This attribute contains the passwords assigned to the object.

- **DS_A_X121_ADDRESS**

This attribute identifies points on the public data network at which the object can be reached or with which it is associated in some other important way. The matching rules require that differences caused by the presence of spaces be considered insignificant.

Selected Object Classes

This section presents the object classes that are defined in the standards. Object classes are groups of directory entries that share certain characteristics. The object classes are arranged into a lattice, based on the object class **DS_O_TOP**. In a lattice, each element, except a leaf, has one or more immediate subordinates but also has one or more immediate superiors. This contrasts with a tree, where each element has exactly one immediate superior. Object classes closer to **DS_O_TOP** are called superclasses, and those further away are called subclasses.

Each directory entry belongs to an object class, and to all the superclasses of that object class. Each entry has an attribute named **DS_A_OBJECT_CLASS**, which was discussed in the previous section, and which identifies the object classes to which the entry belongs. The values of this attribute are object identifiers, which are represented in the interface by constants with the same name as the object class, prefixed by **DS_O_**.

Associated with each object class are zero or more mandatory and zero or more optional attributes. Each directory entry must contain all the mandatory attributes and can (but need not) contain the optional attributes associated with the object class and its superclasses.

The object classes defined in the standards are shown in Table 53 on page 283, together with their object identifiers.

Note: The third and fourth columns of Table 53 on page 283 contain the contents octets of the BER encoding of the object identifier. All these object identifiers stem from the root **{joint-iso-ccitt(2) ds(5) objectClass(6)}**. Basic encoding rules state that the first two decimal numbers be combined according to the formula $x*40+y$ to form the first hexadecimal value. Thus $2.5.4 = 85.4 = \backslashx55\backslashx04$.

Table 53. Object Identifiers for Selected Object Classes

Package	Object Class	Object Identifier BER	
		DECIMAL	HEXADECIMAL
BDCP	DS_O_ALIAS	85.6.1	\x55\x06\x01
BDCP	DS_O_APPLIC_ENTITY	85.6.12	\x55\x06\x0C
BDCP	DS_O_APPLIC_PROCESS	85.6.11	\x55\x06\x0B
BDCP	DS_O_COUNTRY	85.6.2	\x55\x06\x02
BDCP	DS_O_DEVICE	85.6.14	\x55\x06\x0E
BDCP	DS_O_DSA	85.6.13	\x55\x06\x0D
BDCP	DS_O_GROUP_OF_NAMES	85.6.9	\x55\x06\x09
BDCP	DS_O_LOCALITY	85.6.3	\x55\x06\x03
BDCP	DS_O_ORG	85.6.4	\x55\x06\x04
BDCP	DS_O_ORG_PERSON	85.6.7	\x55\x06\x07
BDCP	DS_O_ORG_ROLE	85.6.8	\x55\x06\x08
BDCP	DS_O_ORG_UNIT	85.6.5	\x55\x06\x05
BDCP	DS_O_PERSON	85.6.6	\x55\x06\x06
BDCP	DS_O_RESIDENTIAL_PERSON	85.6.10	\x55\x06\x0A
BDCP	DS_O_TOP	85.6.0	\x55\x06\x00

OM Class Hierarchy

The remainder of this chapter defines the additional OM classes used to represent values of the selected attributes described in “Selected Attribute Types” on page 275. Some of the selected attributes are represented by OM classes that are used in the interface itself, and hence are defined in Chapter 11, “XDS Class Definitions” on page 241, for example, *DS_C_NAME*.

This section shows the hierarchical organization of the OM classes that are defined in the following sections, and it shows which OM classes inherit additional OM attributes from their OM superclasses. In the following list, subclassification is indicated by indentation, and the names of abstract OM classes are in italics. For example, **DS_C_POSTAL_ADDRESS** is an immediate subclass of the abstract OM class *OM_C_OBJECT*.

OM_C_OBJECT

- **DS_C_FACSIMILE_PHONE_NBR**
- **DS_C_POSTAL_ADDRESS**
- **DS_C_SEARCH_CRITERION**
- **DS_C_SEARCH_GUIDE**
- **DS_C_TELETEX_TERM_IDENT**
- **DS_C_TELEX_NBR**

None of the OM classes in the preceding list are encodable using **om_encode()** and **om_decode()**.

DS_C_FACSIMILE_TELEPHONE_NUMBER

An instance of OM class **DS_C_FACSIMILE_PHONE_NBR** identifies and describes a facsimile terminal, if required.

An instance of this OM class has the OM attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attributes listed in Table 54.

Table 54. OM Attributes of *DS_C_FACSIMILE_PHONE_NBR*

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_PARAMETERS	Object(MH_C_G3_FAX_NBPS) ¹	-	0 or 1	-
DS_PHONE_NBR	String(OM_S_PRINTABLE_STRING) ²	1-32	1	-

Notes:

1. As defined in the X.400 API Specifications.
2. As permitted by E.123 (for example, +44 582 10101).

- **DS_PARAMETERS**

If present, this attribute identifies the non-basic capabilities of the facsimile terminal.

- **DS_PHONE_NBR**

This attribute contains a telephone number by which the facsimile terminal is accessed.

DS_C_POSTAL_ADDRESS

An instance of OM class **DS_C_POSTAL_ADDRESS** is a postal address.

An instance of this OM class has the OM attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attributes listed in Table 55.

Table 55. OM Attributes of *DS_C_POSTAL_ADDRESS*

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_POSTAL_ADDRESS	String(OM_S_TELETEX_STRING)	1-30	1-6	-

- **DS_POSTAL_ADDRESS**

Each value of this OM attribute is one line of the postal address. It typically includes a name, street address, city name, state or province name, postal code, and possibly a country name.

DS_C_SEARCH_CRITERION

An instance of OM class **DS_C_SEARCH_CRITERION** is a component of a **DS_C_SEARCH_GUIDE** OM object.

An instance of this OM class has the OM attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attributes listed in Table 56.

Table 56. OM Attributes of *DS_C_SEARCH_CRITERION*

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_ATTRIBUTE_TYPE	String(OM_S_OBJECT_IDENTIFIER_STRING)	-	0 or 1	-
DS_CRITERIA	Object(DS_C_SEARCH_CRITERION)	-	0 or more	-
DS_FILTER_ITEM_TYPE	Enum(DS_Filter_Item_Type)	-	0 or 1	-
DS_FILTER_TYPE	Enum(DS_Filter_Type)	-	1	DS_ITEM

A **DS_C_SEARCH_CRITERION** suggests how to build part of a filter to be used when searching the directory. Its meaning depends on the value of its OM attribute **DS_FILTER_TYPE**. If the value is **DS_ITEM**, the criterion suggests building an instance of OM class **DS_C_FILTER_ITEM**. If **DS_FILTER_TYPE** has any other value, it suggests building an instance of OM class **DS_C_FILTER**.

- **DS_ATTRIBUTE_TYPE**

This attribute indicates the attribute type to be used in the suggested **DS_C_FILTER_ITEM**. This OM attribute is only present when the value of **DS_FILTER_TYPE** is **DS_ITEM**.

- **DS_CRITERIA**

This attribute contains nested search criteria. This OM attribute is not present when the value of **DS_FILTER_TYPE** is **DS_ITEM**.

- **DS_FILTER_ITEM_TYPE**

This attribute indicates the type of suggested filter item. Its value can be one of the following:

- **DS_APPROXIMATE_MATCH**
- **DS_EQUALITY**
- **DS_GREATER_OR_EQUAL**
- **DS_LESS_OR_EQUAL**
- **DS_SUBSTRINGS**

The filter item cannot have the value **DS_PRESENT**. This OM attribute is only present when the value of **DS_FILTER_TYPE** is **DS_ITEM**.

- **DS_FILTER_TYPE**

This attribute indicates the type of suggested filter. The value **DS_ITEM** means that the suggested component is a filter item, not a filter. The other values suggest the corresponding type of filter. Its value is one of the following:

- **DS_AND**
- **DS_ITEM**
- **DS_NOT**

DS_C_SEARCH_GUIDE

An instance of OM class **DS_C_SEARCH_GUIDE** suggests a criterion for searching the Directory for particular entries. It can be used to build a **DS_C_FILTER** for **ds_search()** operations that are based on the object in whose entry the search guide occurs.

An instance of this OM class has the OM attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attributes listed in Table 57.

Table 57. OM Attributes of DS_C_SEARCH_GUIDE

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_OBJECT_CLASS	String(OM_S_OBJECT_IDENTIFIER_STRING)	-	0 or 1	-
DS_CRITERIA	Object(DS_C_SEARCH_CRITERION)	-	1	-

- **DS_OBJECT_CLASS**

This attribute identifies the object class of the entries to which the search guide applies. If this OM attribute is absent, the search guide applies to objects of any class.

- **DS_CRITERIA**

This attribute contains the suggested search criteria.

DS_C_TELETEX_TERMINAL_IDENTIFIER

An instance of OM class **DS_C_TELETEX_TERM_IDENT** identifies and describes a teletex terminal.

An instance of this OM class has the OM attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attributes listed in Table 58.

Table 58. OM Attributes of DS_C_TELETEX_TERM_IDENT

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_PARAMETERS	Object(MH_C_TELETEX_NBPS) ¹	-	0 or 1	-
DS_TELETEX_TERM	String(OM_S_PRINTABLE_STRING) ²	1-1024	1	-

Notes:

1. As defined in the X.400 API Specifications.
2. As permitted by F.200.

- **DS_PARAMETERS**

This attribute identifies the nonbasic capabilities of the teletex terminal.

- **DS_TELETEX_TERMINAL**

This attribute identifies the teletex terminal.

DS_C_TELEX_NUMBER

An instance of OM class **DS_C_TELEX_NBR** identifies and describes a telex terminal.

An instance of this OM class has the OM attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attributes listed in Table 59.

Table 59. OM Attributes of DS_C_TELEX_NBR

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_ANSWERBACK	String(OM_S_PRINTABLE STRING)	1-8	1	-
DS_COUNTRY_CODE	String(OM_S_PRINTABLE STRING)	1-4	1	-
DS_TELEX_NBR	String(OM_S_PRINTABLE STRING)	1-14	1	-

- **DS_ANSWERBACK**

This attribute contains the code with which the telex terminal acknowledges calls placed to it.

- **DS_COUNTRY_CODE**

This attribute contains the identifier of the country through which the telex terminal is accessed.

- **DS_TELEX_NBR**

This attribute contains the number by means of which the telex terminal is addressed.

Chapter 13. Strong Authentication Package

This chapter describes the Strong Authentication Package (SAP). In addition to the attribute types, attribute syntaxes, and object classes defined in the Basic Directory Contents Package, the standards also contain definitions to support authentication mechanisms.² They include such objects as

Strong-Authentication-User.

Note: z/OS DCE does not support GDS. The GDS naming information presented is intended to increase your understanding of DCE name structure and concepts.

This chapter outlines names for each of these items, and it defines OM classes to represent those that are not represented directly by OM syntaxes. The values of attributes in the directory are not restricted to those discussed in this chapter, and new attribute types and syntaxes can be created at any time. (For further information on how the values of other syntaxes are represented in the interface, see “Attribute and Attribute Value Assertion” on page 237.)

The constants and OM classes in this chapter are defined in addition to those in Chapter 11, “XDS Class Definitions” on page 241, since they are not essential to the working of the interface, but instead allow directory entries to be utilized. The definitions belong to the Strong Authentication Package (SAP), which is supported by the DCE XDS API following negotiation of its use with **ds_version()**.

The object identifier associated with the SA Package is **{iso(1) identified-organization(3) icd-ecma(0012) member-company(2) dec(1011) xopen(28) sap(2)}** with the following encoding:

```
\x2B\x0C\x02\x87\x73\x1C\x02
```

This identifier is represented by the constant **DS_STRONG_AUTHENT_PKG**. The C constants associated with this package are in the **xdssap.h** header file.

The concepts and notation used are introduced in “Introduction to OM Classes” on page 241. They are also fully explained in these chapters:

- Chapter 17, “Information Syntaxes” on page 325
- Chapter 18, “XOM Service Interface” on page 331
- Chapter 19, “Object Management Package” on page 347

SAP Attribute Types

This section presents the additional attribute types defined in the standards that are to be used with the Strong Authentication Package. Each attribute type has an object identifier, which is the value of the OM attribute **DS_ATTRIBUTE_TYPE**. These object identifiers are represented in the interface by constants with the same name as the directory attribute, and they are prefixed with **DS_A_** so that they can be easily identified.

This section contains two tables that are used to indicate the object identifiers for Strong Authentication Package attribute types (see Table 60 on page 290), and the values for Strong Authentication Package attribute types (see Table 61 on page 290), respectively. Following these two tables is a brief description of each attribute. (See “Selected Attribute Types” on page 275 for information on general matching rules.)

²

These definitions are chiefly in *The Directory: Selected Attribute Types* (ISO 9594-6, CCITT X.520) and *The Directory: Selected Object Classes* (ISO 9594-7, CCITT X.521) with additional material in *The Directory: Overview of Concepts, Models, and Services* (ISO 9594-1, CCITT X.500) and *The Directory: Authentication Framework* (ISO 9594-8, CCITT X.509).

Note: The third and fourth columns of Table 60 on page 290 contain the contents octets of the BER encoding of the object identifier. All these object identifiers stem from the root **{joint-iso-ccitt(2) ds(5) attributeType(4)}**. Basic encoding rules state that the first two decimal numbers be combined according to the formula $x*40+y$ to form the first hexadecimal value. Thus $2.5.4 = 85.4 = \backslashx55\backslashx04$.

Table 60. Object Identifiers for SAP Attribute Types

Package	Attribute Type	Object Identifier BER	
		DECIMAL	HEXADECIMAL
SAP	DS_A_AUTHORITY_REVOC_LIST	85.4.38	\x55\x04\x26
SAP	DS_A_CA_CERT	85.4.37	\x55\x04\x25
SAP	DS_A_CERT_REVOC_LIST	85.4.39	\x55\x04\x27
SAP	DS_A_CROSS_CERT_PAIR	85.4.40	\x55\x04\x28
SAP	DS_A_USER_CERT	85.4.36	\x55\x04\x24

Table 61. Representation of Values for SAP Attribute Types

Attribute Type	OM Value Syntax	Value Length	Multi-Valued	Matching Rules
DS_A_AUTHORITY_REVOC_LIST	Object(DS_C_CERT_LIST)	-	yes	-
DS_A_CA_CERT	Object(DS_C_CERT)	-	yes	-
DS_A_CERT_REVOC_LIST	Object(DS_C_CERT_LIST)	-	yes	-
DS_A_CROSS_CERT_PAIR	Object(DS_C_CERT_PAIR)	-	yes	-
DS_A_USER_CERT	Object(DS_C_CERT)	-	yes	-

Throughout the descriptions that follow, the term *object* indicates the directory object whose directory entry contains the corresponding directory attributes.

- **DS_A_AUTHORITY_REVOC_LIST**

This attribute occurs only in entries that describe a Certification Authority (CA). It lists all the certificates issued to any of the CAs known to this CA, and later revoked. Each value of this OM attribute is signed by the CA.

- **DS_A_CA_CERT**

This attribute specifies the certificates assigned to the object, which is a Certification Authority.

- **DS_A_CERT_REVOC_LIST**

This attribute occurs only in entries that describe a CA. It lists the certificates issued by this CA and later revoked. Each value of this OM attribute is signed by the CA.

- **DS_A_CROSS_CERT_PAIR**

This attribute specifies one or two certificates, held in the entry of a CA. The first certificate is that of one CA, guaranteed by a second CA; whereas, the second certificate is that of the second CA, guaranteed by the first CA.

- **DS_A_USER_CERT**

This attribute specifies the user certificates assigned to the object, which may be any user certificate including a CA certificate.

Strong Authentication Package Object Classes

This section presents the Strong Authentication Package object classes that are defined in the standards. (See Table 62.)

Note: The third and fourth columns of Table 62 contain the contents octets of the BER encoding of the object identifier. All these object identifiers stem from the root **{joint-iso-ccitt(2) ds(5) objectClass(6)}**.

Table 62. Object Identifiers for SAP Object Classes

Package	Object Class	Object Identifier BER	
		DECIMAL	HEXADECIMAL
SAP	DS_O_CERT_AUTHORITY	85.6.16	\x55\x06\x10
SAP	DS_O_STRONG_AUTHENT_USER	85.6.15	\x55\x06\x0F

OM Class Hierarchy

The remainder of this chapter defines the additional OM classes used by SAP. This section shows the hierarchical organization of the OM classes that are defined in the following sections, and it shows which OM classes inherit additional OM attributes from their OM superclasses. In the following list, subclassification is indicated by indentation, and the names of abstract OM classes are in italics.

OM_C_OBJECT

- **DS_C_ALGORITHM_IDENT**
- **DS_C_CERT_PAIR**
- *DS_C_SIGNATURE*
 - **DS_C_CERT**
 - **DS_C_CERT_LIST**
 - **DS_C_CERT_SUBLIST**

None of the OM classes in the preceding list are encodable by using **om_encode()** and **om_decode()**.

DS_C_ALGORITHM_IDENT

An instance of OM class **DS_C_ALGORITHM_IDENT** records the encryption algorithm that an object uses to digitally sign messages, together with the parameters of the algorithm.

An instance of this OM class has the OM attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attributes listed in Table 63.

Table 63. OM Attributes of *DS_C_ALGORITHM_IDENT*

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_ALGORITHM	String(OM_S_OBJECT_IDENTIFIER_STRING)	-	1	-
DS_ALGORITHM_PARAMETERS	Any	-	0 or 1	-

- **DS_ALGORITHM**

This attribute specifies an object identifier that uniquely identifies the algorithm used by some object.

- **DS_ALGORITHM_PARAMETERS**

This attribute specifies the values of the algorithm's parameters that are used by the object. The syntax of the parameters is determined by each individual algorithm.

DS_C_CERT

An instance of OM class **DS_C_CERT** comprises a user's DN, public key, and additional information, all of which is digitally signed by the issuing CA in order to make the certificate unforgeable. The OM attributes associated with *DS_C_SIGNATURE* (a superclass of **DS_C_CERT**) are present.

An instance of this OM class has the OM attributes of its superclasses, *OM_C_OBJECT* and *DS_C_SIGNATURE*, in addition to the OM attributes listed in Table 64.

Table 64. OM Attributes of *DS_C_CERT*

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_SERIAL_NUMBER	OM_S_INTEGER	-	1	-
DS_SUBJECT	Object(<i>DS_C_NAME</i>)	-	1	-
DS_SUBJECT_ALGORITHM	Object(DS_C_ALGORITHM_IDENT)	-	1	-
DS_SUBJECT_SUBJECT_PUBLIC_KEY	String(OM_S_BIT_STRING)	-	1	-
DS_VALIDITY_NOT_AFTER	String(OM_S_UTC_TIME_STRING)	0-17	1	-
DS_VALIDITY_NOT_BEFORE	String(OM_S_UTC_TIME_STRING)	0-17	1	-
DS_VERSION	Enum(DS_Version)	-	1	DS_V1988

- **DS_SERIAL_NUMBER**

This attribute distinguishes the certificate from all other certificates that have been or will be issued by the CA which issued this certificate.

- **DS_SUBJECT**

This attribute specifies the subject's name.

- **DS_SUBJECT_ALGORITHM**

This attribute specifies the algorithm that is used by the subject and is associated with the public key.

- **DS_SUBJECT_PUBLIC_KEY**

This attribute specifies the subject's public key associated with the algorithm.

- **DS_VALIDITY_NOT_AFTER**

This attribute specifies the last day on which the certificate is valid.

- **DS_VALIDITY_NOT_BEFORE**

This attribute specifies the first day on which the certificate is valid.

- **DS_VERSION**

This attribute identifies the certificate's design. Its value is as follows:

- **DS_V1988**, meaning the design specified in the 1988 version of the standards.

DS_C_CERT_LIST

An instance of OM class **DS_C_CERT_LIST** documents the revocation of zero or more certificates. The documentation is provided by the object, which is a CA whose signature is affixed to the instance.

An instance of this OM class has the OM attributes of its superclasses, *OM_C_OBJECT* and *DS_C_SIGNATURE*, in addition to the OM attributes listed in Table 65.

Table 65. OM Attributes of DS_C_CERT_LIST

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_LAST_UPDATE	String(OM_S_UTC_TIME_STRING)	0-17	1	-
DS_REVOKED_CERTS	Object(DS_C_CERT_SUBLIST)	-	0 or more	-

- **DS_LAST_UPDATE**

This attribute indicates the time at which the revocation list was updated to its current state.

- **DS_REVOKED_CERTS**

This attribute identifies the revoked certificates.

DS_C_CERT_PAIR

An instance of OM class **DS_C_CERT_PAIR** contains one or both of a forward and reverse certificate, that assists users in building a certification path.

An instance of this OM class has the OM attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attributes listed in Table 66.

Table 66. OM Attributes of DS_C_CERT_PAIR

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_FORWARD	Object(DS_C_CERT)	-	0 or 1	-
DS_REVERSE	Object(DS_C_CERT)	-	0 or 1	-

Notes:

1. At least one of these OM attributes must be present.

- **DS_FORWARD**

This attribute specifies the certificate of the first CA which was issued by a second CA.

- **DS_REVERSE**

This attribute specifies the certificate of the second CA which was issued by the first CA.

DS_C_CERT_SUBLIST

An instance of OM class **DS_C_CERT_SUBLIST** documents the revocation of zero or more certificates issued by the CA whose signature is affixed to the instance.

An instance of this OM class has the OM attributes of its superclasses, *OM_C_OBJECT* and *DS_C_SIGNATURE*, in addition to the OM attributes listed in Table 67.

Table 67. OM Attributes of *DS_C_CERT_SUBLIST*

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_REVOCATION_DATE	String(OM_S_UTC_TIME_STRING)	0-17	0 or more	-
DS_SERIAL_NUMBERS	OM_S_INTEGER	-	0 or more	-

Notes:

1. The values of these two OM attributes parallel one another and shall be equal in number.

- **DS_REVOCATION_DATE**

This attribute specifies the epoch at which each of the certificates was revoked. The serial numbers of the certificates are the corresponding values of the OM attribute **DS_SUBJECT** of a corresponding **OS_C_CERT** OM object.

- **DS_SERIAL_NUMBERS**

This attribute specifies the serial numbers assigned to the revoked certificates.

DS_C_SIGNATURE

An instance of the abstract OM class *DS_C_SIGNATURE* contains the algorithm identifier used to produce a digital signature and the name of the object that produced it. The scope of the signature is any instance of any subclass of this OM class.

An instance of this OM class has the OM attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attributes listed in Table 68.

Table 68. OM Attributes of *DS_C_SIGNATURE*

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_ISSUER	Object(<i>DS_C_NAME</i>)	-	1	-
DS_SIGNATURE	Object(DS_C_ALGORITHM_INDENT)	-	1	-
DS_SIGNATURE_VALUE	String(OM_S_OCTET_STRING)	-	1	-

- **DS_ISSUER**

This attribute indicates the name of the object that produced the digital signature.

- **DS_SIGNATURE**

This attribute identifies the algorithm that was used to produce the digital signature, and any parameters of the algorithm.

- **DS_SIGNATURE_VALUE**

An enciphered summary of the information to which the signature is appended. The summary is produced by means of a one-way hash function, while the enciphering is carried out by using the secret key of the signer.

Chapter 14. MHS Directory User Package

The Message Handling Systems Directory User Package (MDUP) contains definitions to support the use of the directory in accordance with the 1988 X.400 User Agents and Message Transfer Agents (MTAs) for name resolution, Distribution List (DL) expansion, and capability assessment. The definitions are based upon the attribute types and syntaxes specified in *X.402, Annex A*.

Note: z/OS DCE does not support GDS. The GDS naming information presented is intended to increase your understanding of DCE name structure and concepts.

The MDUP is an optional package that can be used by the XDS interface. Applications must negotiate use of this package with **ds_version()** before using any of the MDUP features. If an application attempts to use features specific to the package without first negotiating its use, an appropriate error (for example, **OM_NO_SUCH_CLASS**) is returned by the **Object Management** (OM) function.

The object identifier associated with the MDUP is **{iso(1) identified-organization(3) icd-ecma(0012) member-company(2) dec(1011) xopen(28) mdup(3)}** with the following encoding:

```
\x2B\x0C\x02\x87\x73\x1C\x03
```

This identifier is represented by the constant **DS_MHS_DIR_USER_PKG**. The C constants associated with this package are defined in the **xdsmdup.h**, **xmhp.h**, and **xmsga.h** header files (see the *z/OS DCE Application Development Reference*).

The concepts and notation used are first mentioned in Chapter 11, “XDS Class Definitions” on page 241. They are also fully explained in Chapter 17, “Information Syntaxes” on page 325 through Chapter 19, “Object Management Package” on page 347. The attribute types are introduced first, followed by the object classes. Next, the OM class hierarchy and OM class definitions required to support the new attribute types are described.

MDUP Attribute Types

This section presents additional directory attribute types that are used with the MDUP. Each attribute type has an object identifier, which is the value of the OM attribute **DS_ATTRIBUTE_TYPE**. These object identifiers are represented in the interface by constants with the same name as the directory attribute and are prefixed by **DS_A_** so that they can be easily identified.

This section contains two tables that are used to indicate the object identifiers for MDUP attribute types (see Table 69), and the values for MDUP attribute (see Table 70 on page 298) types respectively. Following these two tables is a brief description of each attribute. (See Chapter 12, “Basic Directory Contents Package” on page 275 for information on general matching rules.)

Note: The third and fourth columns of Table 69 contain the contents octets of the BER encoding of the object identifier. All these object identifiers stem from the root **{joint-iso-ccitt(2) mhs-motis(6) arch(5) at(2)}**. Basic encoding rules state that the first two decimal numbers be combined according to the formula $x*40+y$ to form the first hexadecimal value. Thus $2.5.4 = 85.4 = \text{\x55\x04}$.

Table 69 (Page 1 of 2). Object Identifiers for MDUP Attribute Types

Package	Attribute Type	Object Identifier BER	
		DECIMAL	HEXADECIMAL
MDUP	DS_A_DELIV_CONTENT_LENGTH	86.5.2.0	\x56\x05\x02\x00

Table 69 (Page 2 of 2). Object Identifiers for MDUP Attribute Types

Package	Attribute Type	Object Identifier BER	
		DECIMAL	HEXADECIMAL
MDUP	DS_A_DELIV_CONTENT_TYPES	86.5.2.1	\x56\x05\x02\x01
MDUP	DS_A_DELIV_EITS	86.5.2.2	\x56\x05\x02\x02
MDUP	DS_A_DL_MEMBERS	86.5.2.3	\x56\x05\x02\x03
MDUP	DS_A_DL_SUBMIT_PERMS	86.5.2.4	\x56\x05\x02\x04
MDUP	DS_A_MESSAGE_STORE	86.5.2.5	\x56\x05\x02\x05
MDUP	DS_A_OR_ADDRESSES	86.5.2.6	\x56\x05\x02\x06
MDUP	DS_A_PREF_DELIV_METHODS	86.5.2.7	\x56\x05\x02\x07
MDUP	DS_A_SUPP_AUTO_ACTIONS	86.5.2.8	\x56\x05\x02\x08
MDUP	DS_A_SUPP_CONTENT_TYPES	86.5.2.9	\x56\x05\x02\x09
MDUP	DS_A_SUPP_OPT_ATTRIBUTES	86.5.2.10	\x56\x05\x02\x0A

Table 70. Representation of Values for MDUP Attribute Types

Attribute Type	OM Value Syntax	Value Length	Multi-Valued	Matching Rules
DS_A_DELIV_CONTENT_LENGTH	INTEGER	-	No	-
DS_A_DELIV_CONTENT_TYPES	String(OM_S_OBJECT_IDENTIFIER_STRING)	-	Yes	-
DS_A_DELIV_EITS	String(OM_S_OBJECT_IDENTIFIER_STRING)	-	Yes	-
DS_A_DL_MEMBERS	Object(DS_C_OR_NAME)	-	Yes	-
DS_A_DL_SUBMIT_PERMS	Object(DS_C_DL_SUBMIT_PERMS)	-	Yes	-
DS_A_MESSAGE_STORE	Object(DS_C_DS_DN)	-	No	-
DS_A_OR_ADDRESSES	Object(MH_C_OR_ADDRESS)	-	Yes	-
DS_A_PREF_DELIV_METHODS	Enum(MH_Delivery_Mode)	-	No	E
DS_A_SUPP_AUTO_ACTIONS	String(OM_S_OBJECT_IDENTIFIER_STRING)	-	Yes	-
DS_A_SUPP_CONTENT_TYPES	String(OM_S_OBJECT_IDENTIFIER_STRING)	-	yes	-
DS_A_SUPP_OPT_ATTRIBUTES	String(OM_S_OBJECT_IDENTIFIER_STRING)	-	yes	-

Throughout the descriptions that follow, the term *object* indicates the directory object whose directory entry contains the corresponding directory attributes.

- **DS_A_DELIV_CONTENT_LENGTH**

This attribute identifies the maximum content length of the messages whose delivery a user will accept.

- **DS_A_DELIV_CONTENT_TYPES**

This attribute identifies the content types of the messages whose delivery a user will accept.

- **DS_A_DELIV_EITS**

This attribute identifies the Encoded Information Types (EITs) of the messages whose delivery a user will accept.

- **DS_A_DL_MEMBERS**

This attribute identifies the members of a DL.

- **DS_A_DL_SUBMIT_PERMS**

This attribute identifies the users and DLs that may submit messages to a DL.

- **DS_A_MESSAGE_STORE**

This attribute identifies a user's Message Store (MS) by name.

- **DS_A_OR_ADDRESSES**

This attribute specifies a user's or DL's Originator/Recipient (O/R) addresses.

- **DS_A_PREF_DELIV_METHODS**

This attribute identifies, in the order of decreasing preference, the methods of delivery a user prefers.

- **DS_A_SUPP_AUTO_ACTIONS**

This attribute identifies the automatic actions that an MS fully supports.

- **DS_A_SUPP_CONTENT_TYPES**

This attribute identifies the content types of the messages whose syntax and semantics an MS fully supports.

- **DS_A_SUPP_OPT_ATTRIBUTES**

This attribute identifies the optional attributes that an MS fully supports.

MDUP Object Classes

There are five MDUP object classes, and their associated object identifiers (see Table 71).

Note: The third and fourth columns of Table 71 contain the contents octets of the BER encoding of the object identifier. MDUP object identifiers stem from the root **{joint-iso-ccitt(2) mhs-motis(6) arch(5) oc(1)}**. Basic encoding rules state that the first two decimal numbers be combined according to the formula $x*40+y$ to form the first hexadecimal value. Thus $2.5.4 = 85.4 = \backslashx55\backslashx04$.

Table 71. Object Identifiers for MDUP Object Classes

Package	Object Class	Object Identifier BER	
		DECIMAL	HEXADECIMAL
MDUP	DS_O_MHS_DISTRIBUTION_LIST	86.5.1.0	\x56\x05\x01\x00
MDUP	DS_O_MHS_MESSAGE_STORE	86.5.1.1	\x56\x05\x01\x01
MDUP	DS_O_MHS_MESSAGE_TRANS_AG	86.5.1.2	\x56\x05\x01\x02
MDUP	DS_O_MHS_USER	86.5.1.3	\x56\x05\x01\x03
MDUP	DS_O_MHS_USER_AG	86.5.1.4	\x56\x05\x01\x04

MDUP OM Class Hierarchy

The remainder of this chapter defines the additional OM classes used by MDUP. This section shows the hierarchical organization of the OM classes that are defined in the following sections, and shows which classes inherit additional OM attributes from their OM superclasses. In the following list, subclassification is indicated by indentation and the names of abstract OM classes are represented in italic font.

- *MH_C_OR_ADDRESS*
 - **MH_C_OR_NAME**
- **DS_C_DL_SUBMIT_PERMS**

None of the OM classes in the preceding list are encodable using **om_encode()** and **om_decode()**.

MH_C_OR_ADDRESS

An instance of class **MH_C_OR_ADDRESS** distinguishes one user or DL from another, and identifies its point of access to the Message Transfer System (MTS). Every user or DL is assigned one or more MTS access points and thus one or more originator/recipient (O/R) addresses.

The attributes specific to this class are listed in Table 72. The 1988 column indicates that the attribute applies only to the 1988 standard.

Table 72 (Page 1 of 3). Attributes Specific to *MH_C_OR_ADDRESS*

Attribute	Value Syntax	Value Length	Value Number	1988?
MH_T_ADMD_NAME ¹	String(PRINTABLE_STRING)	0-16	0 or 1	-
MH_T_COMMON_NAME	String(OM_S_PRINTABLE_STRING) or String(OM_S_TELETEX_STRING) ²	1-64	0-2	1988
MH_T_COUNTRY_NAME ¹	String(OM_S_PRINTABLE_STRING)	2-3	0 or 1	-
MH_T_DOMAIN_TYPE_1	String(OM_S_PRINTABLE_STRING) or String(OM_S_TELETEX_STRING) ^{2,3}	1-8	0-2 ⁴	-
MH_T_DOMAIN_TYPE_2	String(OM_S_PRINTABLE_STRING) or String(OM_S_TELETEX_STRING) ^{2,3}	1-8	0-2 ⁴	-
MH_T_DOMAIN_TYPE_3	String(OM_S_PRINTABLE_STRING) or String(OM_S_TELETEX_STRING) ^{2,3}	1-8	0-2 ⁴	-
MH_T_DOMAIN_TYPE_4	String(OM_S_PRINTABLE_STRING) or String(OM_S_TELETEX_STRING) ^{2,3}	1-8	0-2 ⁴	-
MH_T_DOMAIN_VALUE_1	String(OM_S_PRINTABLE_STRING) or String(OM_S_TELETEX_STRING) ^{2,3}	1-128	0-2 ⁴	-
MH_T_DOMAIN_VALUE_2	String(OM_S_PRINTABLE_STRING) or String(OM_S_TELETEX_STRING) ^{2,3}	1-128	0-2 ⁴	-
MH_T_DOMAIN_VALUE_3	String(OM_S_PRINTABLE_STRING) or String(OM_S_TELETEX_STRING) ^{2,3}	1-128	0-2 ⁴	-
MH_T_DOMAIN_VALUE_4	String(OM_S_PRINTABLE_STRING) or String(OM_S_TELETEX_STRING) ^{2,3}	1-128	0-2 ⁴	-
MH_T_GENERATION	String(OM_S_PRINTABLE_STRING) or String(OM_S_TELETEX_STRING) ^{2,3}	1-3	0-2 ⁴	-

Table 72 (Page 2 of 3). Attributes Specific to MH_C_OR_ADDRESS

Attribute	Value Syntax	Value Length	Value Number	1988?
MH_T_GIVEN_NAME	String(OM_S_PRINTABLE_STRING) or String(OM_S_TELETEX_STRING) ^{2,3}	1-16	0-2 ⁴	-
MH_T_INITIALS	String(OM_S_PRINTABLE_STRING) or String(OM_S_TELETEX_STRING) ^{2,3}	1-5	0-2 ⁴	-
MH_T_ISDN_NUMBER	String(OM_S_NUMERIC_STRING)	1-15	0 or 1	1988
MH_T_ISDN_SUBADDRESS	String(OM_S_NUMERIC_STRING)	1-40	0 or 1 ⁵	1988
MH_T_NUMERIC_USER_IDENTIFIER	String(NUMERIC_STRING)	1-32	0 or 1	-
MH_T_ORGANIZATION_NAME	String(OM_S_PRINTABLE_STRING) or String(OM_S_TELETEX_STRING) ^{2,3}	1-64	0-2 ^{4,6}	-
MH_T_ORGANIZATIONAL_UNIT_NAME_1	String(OM_S_PRINTABLE_STRING) or String(OM_S_TELETEX_STRING) ^{2,3}	1-32	0-2 ⁴	-
MH_T_ORGANIZATIONAL_UNIT_NAME_2	String(OM_S_PRINTABLE_STRING) or String(OM_S_TELETEX_STRING) ^{2,3}	1-32	0-2 ⁴	-
MH_T_ORGANIZATIONAL_UNIT_NAME_3	String(OM_S_PRINTABLE_STRING) or String(OM_S_TELETEX_STRING) ^{2,3}	1-32	0-2 ⁴	-
MH_T_ORGANIZATIONAL_UNIT_NAME_4	String(OM_S_PRINTABLE_STRING) or String(OM_S_TELETEX_STRING) ^{2,3}	1-32	0-2 ⁴	-
MH_T_POSTAL_ADDRESS_DETAILS	String(OM_S_PRINTABLE_STRING) or String(OM_S_TELETEX_STRING) ²	1-30	0-2	1988
MH_T_POSTAL_ADDRESS_IN_FULL	String(OM_S_TELETEX_STRING)	1-185	0 or 1	1988
MH_T_POSTAL_ADDRESS_IN_LINES	String(OM_S_PRINTABLE_STRING)	1-30	0-6	1988
MH_T_POSTAL_CODE	String(OM_S_PRINTABLE_STRING)	1-16	0 or 1	1988
MH_T_POSTAL_COUNTRY_NAME	String(OM_S_PRINTABLE_STRING)	2-3	0 or 1	1988
MH_T_POSTAL_DELIVERY_POINT_NAME	String(OM_S_PRINTABLE_STRING) or String(OM_S_TELETEX_STRING) ²	1-30	0-2	1988
MH_T_POSTAL_DELIV_SYSTEM_NAME	String(OM_S_PRINTABLE_STRING)	1-16	0 or 1	1988
MH_T_POSTAL_GENERAL_DELIV_ADDR	String(OM_S_PRINTABLE_STRING) or String(OM_S_TELETEX_STRING) ²	1-30	0-2	1988
MH_T_POSTAL_LOCALE	String(OM_S_PRINTABLE_STRING) or String(OM_S_TELETEX_STRING) ²	1-30	0-2	1988
MH_T_POSTAL_OFFICE_BOX_NUMBER	String(OM_S_PRINTABLE_STRING) or String(OM_S_TELETEX_STRING) ^{2,3}	1-30	0-2	1988
MH_T_POSTAL_OFFICE_NAME	String(OM_S_PRINTABLE_STRING) or String(OM_S_TELETEX_STRING) ²	1-30	0-2	1988
MH_T_POSTAL_OFFICE_NUMBER	String(OM_S_PRINTABLE_STRING) or String(OM_S_TELETEX_STRING) ^{2,3}	1-30	0-2	1988
MH_T_POSTAL_ORGANIZATION_NAME	String(OM_S_PRINTABLE_STRING) or String(OM_S_TELETEX_STRING) ^{2,3}	1-30	0-2	1988
MH_T_POSTAL_PATRON_DETAILS	String(OM_S_PRINTABLE_STRING) or String(OM_S_TELETEX_STRING) ²	1-30	0-2	1988

Table 72 (Page 3 of 3). Attributes Specific to MH_C_OR_ADDRESS

Attribute	Value Syntax	Value Length	Value Number	1988?
MH_T_POSTAL_PATRON_NAME	String(OM_S_PRINTABLE_STRING) or String(OM_S_TELETEX_STRING) ²	1-30	0-2	1988
MH_T_POSTAL_STREET_ADDRESS	String(OM_S_PRINTABLE_STRING) or String(OM_S_TELETEX_STRING) ²	1-30	0-2	1988
MH_T_PRESENTATION_ADDRESS	Object(DS_C_PRESENTATION_ADDRESS)	-	0 or 1	1988
MH_T_PRMD_NAME ¹	String(OM_S_PRINTABLE_STRING)	1-16	0 or 1	-
MH_T_SURNAME	String(OM_S_PRINTABLE_STRING) or String(OM_S_TELETEX_STRING) ^{2,3}	1-40	0-2 ⁴	-
MH_T_TERMINAL_IDENTIFIER	String(OM_S_PRINTABLE_STRING)	1-24	0 or 1	-
MH_T_TERMINAL_TYPE	Enum(MH_Terminal_Type)	-	0 or 1	1988
MH_T_X121_ADDRESS	String(NUMERIC_STRING)	1-15	0 or 1	-

Notes:

1. The value initially is the current session's attribute of the same name.
2. If only one value is present in international communication, its syntax is String(OM_S_PRINTABLE_STRING). If two values are present, in either domestic or international communication, the syntax of the first is String(OM_S_PRINTABLE_STRING), the syntax of the second is String(OM_S_TELETEX_STRING), and the two convey the same information such that either can be safely ignored. For example, Teletex strings allow inclusion of the accented characters commonly used in many countries. Not all input/output devices, however, permit the entry and display of such characters. Printable strings are required internationally to ensure that such device limitations do not prevent communications.
3. For 1984 the syntax of the value is String(OM_S_PRINTABLE_STRING).
4. For 1984, at most one value shall be present.
5. This attribute is present only if the ISDN Number attribute is present.
6. For 1988, this attribute is required if any Organization Name is present.

• **MH_T_ADMD_NAME**

The name of the user's or DL's Administration Management Domain (ADMD). It identifies the ADMD relative to the country that the **MH_T_COUNTRY_NAME** attribute indicates. Its values are defined by that country.

Note that the attribute value that comprises a single space is reserved. If permitted by the country that the **MH_T_COUNTRY_NAME** attribute indicates, a single space designates "any," that is, all ADMDs within the country. This affects both the identification of users and DLs within the country and the routing of messages, probes, and reports to and among the ADMDs of that country. Regarding the former, it requires that the O/R addresses of users and DLs within the country be chosen so as to ensure they are not ambiguous, even in the absence of the actual names of the users' and DLs' ADMDs. Regarding the latter, it permits both Private Management Domains (PRMD) within, and ADMDs outside the country to route messages, probes, and reports to any of the ADMDs within the country indiscriminately. It also requires that the ADMDs within the country interconnect themselves in such a way that the messages, probes, and reports are conveyed to their destinations.

• **MH_T_COMMON_NAME**

This attribute contains the name commonly used to refer to the user or DL. It identifies the user or DL relative to the entity indicated by another attribute; for example **MH_T_ORGANIZATION_NAME**. Its values are defined by that entity.

- **MH_T_COUNTRY_NAME**

This attribute contains the name of the user's or DL's country. Its defined values are the numbers that X.121 assigns to the country, or the character pairs that ISO 3166 assigns to it.

- **MH_T_DOMAIN_TYPE_1**

This attribute contains the name of a class of information. Its values are defined by the user's or DL's ADMD and PRMD, if any, in combination.

- **MH_T_DOMAIN_TYPE_2**

This attribute contains the name of a class of information. Its values are defined by the user's or DL's ADMD and PRMD, if any, in combination.

- **MH_T_DOMAIN_TYPE_3**

This attribute contains the name of a class of information. Its values are defined by the user's or DL's ADMD and PRMD, if any, in combination.

- **MH_T_DOMAIN_TYPE_4**

This attribute contains the name of a class of information. Its values are defined by the user's or DL's ADMD and PRMD, if any, in combination.

- **MH_T_DOMAIN_VALUE_1**

This attribute is an instance of the class of information that the **MH_T_DOMAIN_TYPE_1** attribute indicates.

- **MH_T_DOMAIN_VALUE_2**

This attribute is an instance of the class of information that the **MH_T_DOMAIN_TYPE_2** attribute indicates.

- **MH_T_DOMAIN_VALUE_3**

This attribute is an instance of the class of information that the **MH_T_DOMAIN_TYPE_3** attribute denotes.

- **MH_T_DOMAIN_VALUE_4**

This attribute is an instance of the class of information that the **MH_T_DOMAIN_TYPE_4** attribute indicates.

- **MH_T_GENERATION**

This attribute contains the user's generation; for example **Jnr**.

- **MH_T_GIVEN_NAME**

This attribute contains the user's given name; for example **Robert**.

- **MH_T_INITIALS**

This attribute contains the initials of all of the user's names except the user's surname; for example **RE**.

- **MH_T_ISDN_NUMBER**

This attribute contains the ISDN number of the user's terminal. Its values are defined by CCITT E.163 and E.164.

- **MH_T_ISDN_SUBADDRESS**

This attribute contains the ISDN subaddress, if any, of the user's terminal. Its values are defined by CCITT E.163 and E.164.

- **MH_T_NUMERIC_USER_IDENTIFIER**

This attribute numerically identifies the user or DL relative to the ADMD that the **MH_T_ADMD_NAME** attribute indicates. Its values are defined by that ADMD.

- **MH_T_ORGANIZATION_NAME**

This attribute contains the name of the user's or DL's organization. As a national matter, such names may be assigned by the country that the **MH_T_COUNTRY_NAME** attribute indicates, the ADMD that the **MH_T_ADMD_NAME** attribute indicates, the PRMD that the **MH_T_PRMD_NAME** attribute indicates, or the latter two organizations together.

- **MH_T_ORGANIZATIONAL_UNIT_NAME_1**

This attribute contains the name of a unit (for example, a division or department) of the organization that the **MH_T_ORGANIZATION_NAME** attribute indicates. The attribute's values are defined by that organization.

- **MH_T_ORGANIZATIONAL_UNIT_NAME_2**

This attribute contains the name of a subunit (for example, a division or department) of the unit that the **MH_T_ORGANIZATIONAL_UNIT_NAME_1** attribute indicates. The attribute's values are defined by the latter unit.

- **MH_T_ORGANIZATIONAL_UNIT_NAME_3**

This attribute contains the name of a subunit (for example, a division or department) of the unit that the **DS_A_ORGANIZATIONAL_UNIT_NAME_2** attribute indicates. The attribute's values are defined by the latter unit.

- **MH_T_ORGANIZATIONAL_UNIT_NAME_4**

This attribute contains the name of a subunit (for example, a division or department) of the unit that the **MH_T_ORGANIZATIONAL_UNIT_NAME_3** attribute indicates. The attribute's values are defined by the latter unit.

- **MH_T_POSTAL_ADDRESS_DETAILS**

This attribute contains the means (for example, room and floor numbers in a large building) for identifying the exact point at which the user takes delivery of physical messages.

- **MH_T_POSTAL_ADDRESS_IN_FULL**

This attribute contains the free-form and possibly multiline postal address of the user as a single Teletex string with the lines being separated as prescribed for Teletex strings.

- **MH_T_POSTAL_ADDRESS_IN_LINES**

This attribute contains the free-form postal address of the user in a sequence of printable strings, each representing a line of text.

- **MH_T_POSTAL_CODE**

This attribute contains the postal code for the geographical area in which the user takes delivery of physical messages. It identifies the area relative to the country that the **MH_T_POSTAL_COUNTRY_NAME** attribute indicates. Its values are defined by the postal administration of that country.

- **MH_T_POSTAL_COUNTRY_NAME**

This attribute contains the name of the country in which the user takes delivery of physical messages. Its defined values are the numbers X.121 assigns to the country, or the character pairs ISO 3166 assigns to it.

- **MH_T_POSTAL_DELIVERY_POINT_NAME**

This attribute identifies the locus of distribution, other than that indicated by the **MH_T_POSTAL_OFFICE_NAME** attribute (for example, a geographical area) of the user's physical messages.

- **MH_T_POSTAL_DELIV_SYSTEM_NAME**

This attribute contains the name of the Postal Delivery System (PDS) through which the user is to receive physical messages. It identifies the PDS relative to the ADMD that the **MH_T_ADMD_NAME** attribute indicates. Its values are defined by that ADMD.

- **MH_T_POSTAL_GENERAL_DELIV_ADDRESS**

This attribute contains the code that the user gives to the post office to collect the physical messages awaiting delivery to the user. The post office is indicated in the **MH_T_POSTAL_OFFICE_NAME** attribute. The values for the **MH_T_POSTAL_GENERAL_DELIV_ADDRESS** attribute are defined by that post office.

- **MH_T_POSTAL_LOCALE**

This attribute identifies the point of delivery, other than that indicated by the following attributes:

- **MH_T_POSTAL_GENERAL_DELIV_ADDR**
- **MH_T_POSTAL_OFFICE_BOX_NUMBER**
- **MH_T_POSTAL_STREET_ADDRESS.**

For example, a building or a hamlet of the user's physical messages.

- **MH_T_POSTAL_OFFICE_BOX_NUMBER**

This attribute contains the number of the post office box by which the user takes delivery of physical messages. The box is located at the post office that the **MH_T_POSTAL_OFFICE_NAME** attribute indicates. The attribute's values are defined by that post office.

- **MH_T_POSTAL_OFFICE_NAME**

This attribute contains the name of the municipality (for example, city or village) where the post office is situated through which the user takes delivery of physical messages.

- **MH_T_POSTAL_OFFICE_NUMBER**

This attribute contains the means of distinguishing among several post offices indicated by the **MH_T_POSTAL_OFFICE_NAME** attribute.

- **MH_T_POSTAL_ORGANIZATION_NAME**

This attribute contains the name of the organization through which the user takes delivery of physical messages.

- **MH_T_POSTAL_PATRON_DETAILS**

This attribute contains additional information (for example, the name of the organizational unit through which the user takes delivery of physical messages) necessary to identify the user for purposes of physical delivery.

- **MH_T_POSTAL_PATRON_NAME**

This attribute contains the name under which the user takes delivery of physical messages.

- **MH_T_POSTAL_STREET_ADDRESS**

This attribute contains the street address (for example, **43 Primrose Lane**) at which the user takes delivery of physical messages.

- **MH_T_PRESENTATION_ADDRESS**

This attribute contains the presentation address of the user's terminal.

- **MH_T_PRMD_NAME**

This attribute contains the name of the user's PRMD. As a national matter, such names may be assigned by the country that the **MH_T_COUNTRY_NAME** attribute indicates or the ADMD that the **MH_T_ADMD_NAME** attribute indicates.

- **MH_T_SURNAME**

This attribute contains the user's surname; for example, **Lee**.

- **MH_T_TERMINAL_IDENTIFIER**

This attribute contains the terminal identifier of the user's terminal; for example, a Telex answer back or a Teletex terminal identifier.

- **MH_T_TERMINAL_TYPE**

This attribute contains the type of the user's terminal. Its value is selected from the following:

- **MH_TT_G3_FAX**
- **MH_TT_G4_FAX**
- **MH_TT_IA5_TERMINAL**
- **MH_TT_TELETEX**
- **MH_TT_TELEX**
- **MH_TT_VIDEOTEX**

The meaning of each value is indicated by its name.

- **X121_ADDRESS**

This attribute contains the network address of the user's terminal. Its values are defined by X.121.

Note: The strings admitted by X.121 include a telephone number preceded by the telephone escape digit (9), and a Telex number preceded by the Telex escape digit (8).

Certain attributes are grouped together for reference as follows:

- **Personal Name attributes**

These comprise the following:

- **MH_T_GIVEN_NAME**
- **MH_T_INITIALS**
- **MH_T_SURNAME**
- **MH_T_GENERATION**

- **Organizational Unit Name attributes**

These comprise the following:

- **MH_T_ORGANIZATIONAL_UNIT_NAME_1**
- **MH_T_ORGANIZATIONAL_UNIT_NAME_2**
- **MH_T_ORGANIZATIONAL_UNIT_NAME_3**
- **MH_T_ORGANIZATIONAL_UNIT_NAME_4**

- **Network Address attributes**

These comprise the following:

- **MH_T_ISDN_NUMBER**
- **MH_T_ISDN_SUBADDRESS**
- **MH_T_PRESENTATION_ADDRESS**
- **MH_T_X121_ADDRESS**

For any *i* in the interval [1, 4], the Domain Type *i* and Domain Value *i* attributes constitute a Domain-Defined Attribute (DDA).

Note: The widespread avoidance of DDAs produces more uniform and thus more user-friendly O/R addresses. However, it is anticipated that not all Management Domains (MD) will be able to avoid such attributes immediately. The purpose of DDAs is to permit an MD to retain its existing native addressing conventions for a time. It is intended, however, that all MDs migrate away from the use of DDAs, and thus that DDAs are used only for an interim period.

An O/R address may take any of the forms summarized in Table 73. Table 73 indicates the attributes that may be present in an O/R address of each form. It also indicates whether it is mandatory (M) or conditional (C) that they do so. When applied to a group of attributes (the network address attributes, for example), mandatory (M) means that at least one member of the group must be present, while conditional (C) means that no members of the group need necessarily be present.

The presence or absence in a particular O/R address of conditional attributes is determined as follows: if a user or DL is accessed through a PRMD, the ADMD that the **MH_T_COUNTRY_NAME** and **MH_T_ADMD_NAME** attributes indicate governs whether attributes used to route messages to the PRMD are present, but it imposes no other constraints on attributes; if a user or DL is *not* accessed through a PRMD, the same ADMD governs whether all conditional attributes, except those specific to postal O/R addresses, are present. All conditional attributes specific to postal O/R addresses are present or absent so as to satisfy the postal addressing requirements of the users they identify.

Table 73 (Page 1 of 2). Forms of Originator/Recipient Address

Attribute	Mnem ¹	Num ²	Spost ³	Upost ⁴	Term ⁵
MH_T_ADMD_NAME	M	M	M	M	C
MH_T_COMMON_NAME	C	-	-	-	-
MH_T_COUNTRY_NAME	M	M	M	M	C
Domain-Defined Attributes	C	C	-	-	C
Network Address Attributes	-	-	-	-	M
MH_T_NUMERIC_USER_IDENTIFIER	-	M	-	-	-
MH_T_ORGANIZATION_NAME	C	-	-	-	-
Organizational Unit Name Attributes	C	-	-	-	-
Personal Name Attributes	C	-	-	-	-
MH_T_POSTAL_ADDRESS_DETAILS	-	-	C	-	-
MH_T_POSTAL_ADDRESS_IN_FULL	-	-	-	M	-
MH_T_POSTAL_CODE	-	-	M	M	-
MH_T_POSTAL_COUNTRY_NAME	-	-	M	M	-
MH_T_POSTAL_DELIVERY_POINT_NAME	-	-	C	-	-
MH_T_POSTAL_DELIV_SYSTEM_NAME	-	-	C	C	-
MH_T_POSTAL_GENERAL_DELIV_ADDR	-	-	C	-	-
MH_T_POSTAL_LOCALE	-	-	C	-	-

Table 73 (Page 2 of 2). Forms of Originator/Recipient Address

Attribute	Mnem ¹	Num ²	Spost ³	Upost ⁴	Term ⁵
MH_T_POSTAL_OFFICE_BOX_NUMBER	-	-	C	-	-
MH_T_POSTAL_OFFICE_NAME	-	-	C	-	-
MH_T_POSTAL_OFFICE_NUMBER	-	-	C	-	-
MH_T_POSTAL_ORGANIZATION_NAME	-	-	C	-	-
MH_T_POSTAL_PATRON_DETAILS	-	-	C	-	-
MH_T_POSTAL_PATRON_NAME	-	-	C	-	-
MH_T_POSTAL_STREET_ADDRESS	-	-	C	-	-
MH_T_PRMD_NAME	C	C ⁶	C	C	C ⁶
MH_T_TERMINAL_IDENTIFIER	-	-	-	-	C
MH_T_TERMINAL_TYPE	-	-	-	-	C

Notes:

1. Mnemonic. X.400 (1984) calls this Form 1 Variant 1.
2. Numeric. X.400 (1984) calls this Form 1 Variant 1.
3. Structured postal. For 1984, this O/R address form is undefined.
4. Unstructured postal. For 1984 this O/R address form is undefined.
5. X.400 (1984) calls this Form 1 Variant 3 and Form 2.
6. For 1984, this attribute is absent(-). For 1988, it is conditional(C).

• **Mnemonic O/R Address**

This address mnemonically identifies a user or DL. Using the **MH_T_ADMD_NAME** and **MH_T_COUNTRY_NAME** attributes, it identifies an ADMD. Using the **MH_T_COMMON_NAME** attribute or the personal name attributes, the **MH_T_ORGANIZATION_NAME** attribute, the Organizational Unit Name attributes, the **MH_T_PRMD_NAME** attribute, or a combination of these, and optionally DDAs, it identifies a user or DL relative to the ADMD.

The personal name attributes identify a user or DL relative to the entity indicated by another attribute; for example, **MH_T_ORGANIZATION_NAME**. The **MH_T_SURNAME** attribute will be present if any of the other three personal name attributes are present.

• **Numeric O/R Address**

This address numerically identifies a user or DL. Using the **MH_T_ADMD_NAME** and **MH_T_COUNTRY_NAME** attributes, it identifies an ADMD. Using the **MH_T_NUMERIC_USER_IDENTIFIER** attribute and possibly the **MH_T_PRMD_NAME** attribute, it identifies the user or DL relative to the ADMD. Any DDAs provide information that is additional to that required to identify the user or DL.

• **Postal O/R Address**

This address identifies a user by means of its postal address. Two kinds of postal O/R address are distinguished:

– **Structured**

Said of a postal O/R address that specifies a user's postal address by means of several attributes. The structure of the postal address is described in the following text in some detail.

– **Unstructured**

Said of a postal O/R address that specifies a user's postal address in a single attribute. The structure of the postal address is left largely unspecified in the following text.

Whether structured or unstructured, a postal O/R address does the following. Using the **MH_T_ADMD_NAME** and **MH_T_COUNTRY_NAME** attributes, it identifies an ADMD. Using the **MH_T_POSTAL_CODE** and **MH_T_POSTAL_COUNTRY_NAME** attributes, it identifies the geographical region in which the user takes delivery of physical messages. Using the **MH_T_POSTAL_DELIV_SYSTEM_NAME** or **MH_T_PRMD_NAME** attribute or both, it also may identify the PDS by means of which the user is to be accessed.

An unstructured postal O/R address also includes the **MH_T_POSTAL_ADDRESS_IN_FULL** attribute. A structured postal O/R address also includes every other postal addressing attribute that the PDS requires to identify the postal patron.

Note: The total number of characters in the values of all attributes, except for **MH_T_ADMD_NAME**, **MH_T_COUNTRY_NAME**, and **MH_T_POSTAL_DELIV_SYSTEM_NAME**, in a postal O/R address should be small enough to permit their rendition in 6 lines of 30 characters, the size of a typical physical envelope window. The rendition algorithm, while defined by the Physical Delivery Access Unit (PDAU), is likely to include inserting delimiters (for example, spaces) between some attribute values.

- **Terminal O/R Address**

This address identifies a user by identifying the user's terminal, using the network address attributes. It also may identify the ADMD through which the terminal is accessed by using the **MH_T_ADMD_NAME** and **MH_T_COUNTRY_NAME** attributes. The **MH_T_PRMD_NAME** attribute and any DDAs, which shall be present only if the **MH_T_ADMD_NAME** and **MH_T_COUNTRY_NAME** attributes are present, provide information additional to that required to identify the user.

If the terminal is a Telematic terminal, it gives the terminal's network address and possibly, using the **MH_T_TERMINAL_TYPE** and **MH_T_TERMINAL_IDENTIFIER** attributes, its terminal type and identifier. If the terminal is a Telex terminal, it gives the terminal's Telex number.

Whenever two O/R addresses are compared for equality, the following differences are ignored:

- Whether an attribute has a value whose syntax is String(**OM_S_PRINTABLE_STRING**), a value whose syntax is String(**OM_S_TELETEX_STRING**), or both.
- Whether a letter in a value of an attribute not used in DDAs is an uppercase or lowercase letter.
- All leading, all trailing, and all but one consecutive embedded spaces in an attribute value.

Note: An MD may impose additional equivalence rules upon the O/R addresses it assigns to its own users and DLs. It may define, for example, rules concerning punctuation characters in attribute values, the case of letters in attribute values, or the relative order of DDAs.

As a national matter, MDs may impose additional rules regarding any attribute that may have a value whose syntax is String(**OM_S_PRINTABLE_STRING**), a value whose syntax is String(**OM_S_TELETEX_STRING**), or both. In particular, the rules for deriving from a Teletex string the equivalent printable string may be nationally prescribed.

MH_C_OR_NAME

An instance of class **MH_C_OR_NAME** comprises a directory name, an O/R address, or both. The name is considered present if, and only if, the **MH_T_DIRECTORY_NAME** attribute is present. The address comprises the attributes specific to the **MH_C_OR_ADDRESS** class and is considered present if, and only if, at least one of those attributes is present.

An O/R name's composition is context sensitive. At submission, the name, the address, or both may be present. At transfer or delivery, the address is present and the name can (but need not) be present. Whether at submission, transfer or delivery, the MTS uses the name, if it is present, only if the address is absent or invalid.

The attributes specific to this class are listed in Table 74.

Table 74. Attribute Specific to MH_C_OR_NAME

Attribute	Value Syntax	Value Length	Value Number	Value Initially	1988?
MH_T_DIRECTORY_NAME	Object(DS_C_NAME)	-	0 or 1	-	1988

- **MH_T_DIRECTORY_NAME**

This attribute contains the name assigned to the user or DL by the worldwide X.500 directory.

DS_C_DL_SUBMIT_PERMS

An instance of OM class **DS_C_DL_SUBMIT_PERMS** characterizes an attribute each of whose values are a submit permission. An instance of this OM class has the OM attributes of its superclass, **OM_C_OBJECT**, and additionally the OM attributes listed in Table 75.

Table 75. OM Attributes of DS_C_DL_SUBMIT_PERMS

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_PERM_TYPE	Enum(DS_Permission_Type)	-	1	-
DS_INDIVIDUAL	Object(MH_C_OR_NAME)	-	0 or 1	-
DS_MEMBER_OF_DL	Object(MH_C_OR_NAME)	-	0 or 1	-
DS_PATTERN_MATCH	Object(MH_C_OR_NAME)	-	0 or 1	-
DS_MEMBER_OF_GROUP	Object(DS_C_DS_DN)	-	0 or more	-

- **DS_PERM_TYPE**

This attribute contains the type of the permission specified herein. Its value can be one of the following:

- **DS_PERM_INDIVIDUAL**
- **DS_PERM_MEMBER_OF_DL**
- **DS_PERM_PATTERN_MATCH**
- **DS_PERM_MEMBER_OF_GROUP**

- **DS_INDIVIDUAL**

This attribute contains the user or unexpanded DL, any of whose O/R names is equal to the specified O/R Name.

- **DS_MEMBER_OF_DL**

This attribute contains each member of the DL, any of whose O/R names is equal to the specified O/R name, or of each nested DL, recursively.

- **DS_PATTERN_MATCH**

This attribute contains each user or unexpanded DL, any of whose O/R names matches the specified O/R name pattern.

- **DS_MEMBER_OF_GROUP**

This attribute contains each member of the group-of-names whose name is specified, or of each nested group-of-names, recursively.

Note that exactly one of the four Name attributes will be present at any time, according to the value of the **DS_PERM_TYPE** attribute.

Chapter 15. Global Directory Service Package

The Global Directory Service Package (GDSP) is an OSF extension to the XDS interface. Applications must negotiate use of this package with **ds_version()** before using any of the additional features. If an application attempts to use features specific to the package without first negotiating its use, then an appropriate error (for example, **OM_NO_SUCH_CLASS**) is returned by the Object Management function.

Note: z/OS DCE does not support GDS. The GDS naming information presented is intended to increase your understanding of DCE name structure and concepts.

The object identifier associated with the GDSP package is **{iso(1)identified-organization(3)icd-ecma(0012) member-company(2) siemens-units(1107) sni(1) directory(3) xdsapi(100) gdsp(0)}** with the following encoding:

```
\x2B\x0C\x02\x88\x53\x01\x03\x64\x00
```

The identifier is represented by the constant **DSX_GDS_PKG**. The C constants associated with this package are contained in the **xdsgds.h** header file (see *z/OS DCE Application Development Reference*).

The concepts and notation used are first mentioned in Chapter 11, “XDS Class Definitions” on page 241. They are also fully explained in Chapter 17, “Information Syntaxes” on page 325 through Chapter 19, “Object Management Package” on page 347. The attribute types are introduced first, followed by the object classes. Next, the OM class hierarchy and OM class definitions required to support the new attribute types are described.

GDSP Attribute Types

This section presents the additional directory attribute types that are used with GDSP. Each attribute type has an object identifier, which is the value of the OM attribute **DS_ATTRIBUTE_TYPE**. These object identifiers are represented in the interface by constants with the same name as the directory attribute, and are prefixed by **DSX_A_** so that they can be easily identified.

This section contains two tables that are used to indicate the object identifiers for GDSP attribute types (see Table 76) and the values for GDSP attribute types (see Table 77 on page 314), respectively. Following these two tables is a brief description of each attribute. (See “Selected Attribute Types” on page 275 for information on general matching rules.)

Table 76 shows the names of the GDSP attribute types, together with the BER encoding of the object identifiers associated with each of them.

Note: The third column of Table 76 contains the contents octets of the BER encoding of the object identifier in hexadecimal. All these object identifiers stem from the root **{iso(1)identified-organization(3) icd-ecma(0012) member-company(2) siemens-units(1107) sni(1) directory(3) attribute-type(4)}**. Basic encoding rules state that the first two decimal numbers be combined according to the formula $x*40+y$ to form the first hexadecimal value. Thus $2.5.4 = 85.4 = \backslashx55\x04$.

Table 76 (Page 1 of 2). Object Identifiers for GDSP Attribute Types

Package	Attribute Type	Object Identifier BER
		HEXADECIMAL
GDSP	DSX_A_ACL	\x2B\x0C\x02\x88\x53\x01\x03\x04\x01
GDSP	DSX_A_AT	\x2B\x0C\x02\x88\x53\x01\x03\x04\x06

Table 76 (Page 2 of 2). Object Identifiers for GDSP Attribute Types

Package	Attribute Type	Object Identifier BER
		HEXADECIMAL
GDSP	DSX_A_CACHE_ATTR	\x2B\x0C\x02\x88\x53\x01\x03\x04\x07
GDSP	DSX_A_CDS_CELL	\x2B\x0C\x02\x88\x53\x01\x03\x04\x0D
GDSP	DSX_A_CDS_REPLICA	\x2B\x0C\x02\x88\x53\x01\x03\x04\x0E
GDSP	DSX_A_CLIENT	\x2B\x0C\x02\x88\x53\x01\x03\x04\x0A
GDSP	DSX_A_DEFAULT_DSA	\x2B\x0C\x02\x88\x53\x01\x03\x04\x08
GDSP	DSX_A_DNLIST	\x2B\x0C\x02\x88\x53\x01\x03\x04\x0B
GDSP	DSX_A_LOCDSA	\x2B\x0C\x02\x88\x53\x01\x03\x04\x09
GDSP	DSX_A_MASTER_KNOWLEDGE	\x2B\x0C\x02\x88\x53\x01\x03\x04\x00
GDSP	DSX_A_OCT	\x2B\x0C\x02\x88\x53\x01\x03\x04\x05
GDSP	DSX_A_SHADOWED_BY	\x2B\x0C\x02\x88\x53\x01\x03\x04\x03
GDSP	DSX_A_SHADOWING_JOB	\x2B\x0C\x02\x88\x53\x01\x03\x04\x0C
GDSP	DSX_A_SRT	\x2B\x0C\x02\x88\x53\x01\x03\x04\x04
GDSP	DSX_A_TIME_STAMP	\x2B\x0C\x02\x88\x53\x01\x03\x04\x02

Table 77 shows the names of the attribute types, together with the OM Value Syntax used in the interface to represent values of that attribute type. The table also includes the range of lengths permitted for the string types, indicates whether the attribute can be multivalued, and lists which matching rules are provided for the syntax.

Table 77. Representation of Values for GDSP Attribute Types

Attribute Type	OM Value Syntax	Value Length	Multi-Valued	Matching Rules
DSX_A_ACL	Object(DSX_C_GDS_ACL)	-	No	E
DSX_A_AT	String(OM_S_PRINTABLE_STRING)	101	Yes	E, S
DSX_A_CACHE_ATTR	No syntax, no values	-	-	-
DSX_A_CDS_CELL	String(OM_S_OCTET_STRING)	36	No	E
DSX_A_CDS_REPLICA	String(OM_S_OCTET_STRING)	45	Yes	E
DSX_A_CLIENT	Only a cache entry	-	-	-
DSX_A_DEFAULT_DSA	Only a cache entry	-	-	-
DSX_A_DNLIST	Object(DS_C_DS_DN)	1K max.	Yes	E, S
DSX_A_LOCDSA	Only a cache entry	-	-	-
DSX_A_MASTER_KNOWLEDGE	Object(DS_C_DS_DN)	1K max.	No	E, S
DSX_A_OCT	String(OM_S_PRINTABLE_STRING)	310	Yes	E, S
DSX_A_SHADOWED_BY	Not used yet	-	-	-
DSX_A_SHADOWING_JOB	Not used yet	-	-	-
DSX_A_SRT	String(OM_S_PRINTABLE_STRING)	56	Yes	E, S
DSX_A_TIME_STAMP	String(OM_S_UTC_TIME_STRING)	18	No	E, 0

Note: With the exception of the **DSX_A_ACL** attribute, the GDSP attributes in the Table 77 are only to be manipulated through the GDS administration interface

Descriptions of the GDS attributes follow:

- **DSX_A_ACL**

This attribute describes the access rights for one or more Directory Service users.

- **DSX_A_AT**

This attribute describes the attribute types permitted in GDS.

- **DSX_A_CACHE_ATTR**

This attribute is used internally by GDS to separate return values that can be cached from those that cannot be cached.

- **DSX_A_CDS_CELL** and **DSX_A_CDS_REPLICA**

These two attributes always exist together in the same object. They describe the information necessary for contacting a remote CDS cell.

- **DSX_A_CLIENT**

This attribute is a cache entry. This naming attribute allows the DUA to retrieve its own PSAP address.

- **DSX_A_DEFAULT_DSA**

This attribute is a cache entry. This naming attribute allows the DUA to retrieve the PSAP address of its default DSA.

- **DSX_A_DNLIST**

This attribute is used internally by the GDS DSA.

- **DSX_A_LOCDSA**

This attribute is a cache entry. This naming attribute allows the DSA to retrieve its own PSAP address.

- **DSX_A_MASTER_KNOWLEDGE**

This attribute contains the Distinguished Name (DN) of the DSA that holds the master copy of this entry.

- **DSX_A_OCT**

This attribute describes the object classes supported by the GDS DSA.

- **DSX_A_SHADOWED_BY** and **DSX_A_SHADOWING_JOB**

These two GDSP attributes are intended for future use.

- **DSX_A_SRT**

This attribute describes the structure of the DNs permitted in GDS.

- **DSX_A_TIME_STAMP**

This attribute is part of the **DSX_O_SCHEMA** object. It contains the creation time of the **DSX_O_SCHEMA** object.

GDSP Object Classes

The only additional GDSP object class is **DSX_O_SCHEMA** (see Table 78). It is stored in GDS as an object directly under root. The most important attributes of the **DSX_O_SCHEMA** object are the three recurring attributes **DSX_A_OCT**, **DSX_A_AT**, and **DSX_A_SRT**. These three objects describe the GDS DIT structure.

Note: The third column of Table 78 contains the contents octets of the BER encoding of the object identifier (in hex). This object identifier stems from the root **{iso(1) identified-organization(3) idc-ecma(12) member-company(2) siemens-units(1107) sni(1) directory(3) object-class(6)}**.. Basic encoding rules state that the first two decimal numbers be combined according to the formula $x*40+y$ to form the first hexadecimal value. Thus $2.5.4 = 85.4 = \backslashx55\backslashx04$.

Table 78. Object Identifiers for GDSP Object Classes

Package	Object Class	Object Identifier BER
		HEXADECIMAL
GDSP	DSX_O_SCHEMA	\x2B\x0C\x02\x88\x53\x01\x03\x06\x00

GDSP OM Class Hierarchy

The remainder of this chapter defines the additional OM classes used by GDSP. This section shows the hierarchical organization of the OM classes that are defined in the following sections, and it shows which classes inherit additional OM attributes from their OM superclasses. In the following list, subclassification is indicated by indentation, and the names of abstract OM classes are represented in italics.

OM_C_OBJECT (defined in the OM package)

- **DS_C_SESSION** (defined in the Directory Service Package)
 - **DSX_C_GDS_SESSION**
- **DS_C_CONTEXT** (defined in the Directory Service Package)
 - **DSX_C_GDS_CONTEXT**
- **DSX_C_GDS_ACL**
- **DSX_C_GDS_ACL_ITEM**

None of the OM classes in the preceding list are encodable using **om_encode()** and **om_decode()**.

DSX_C_GDS_ACL

An instance of OM class **DSX_C_GDS_ACL** describes up to five categories of rights for one or more directory users.

An instance of this OM class has the OM attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attributes listed in Table 79.

Table 79 (Page 1 of 2). OM Attributes of DSX_C_GDS_ACL

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DSX_MODIFY_PUBLIC	Object(DSX_C_GDS_ACL_ITEM)	-	1 - 4	-

Table 79 (Page 2 of 2). OM Attributes of DSX_C_GDS_ACL

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DSX_READ_STANDARD	Object(DSX_C_GDS_ACL_ITEM)	-	1 - 4	-
DSX_MODIFY_STANDARD	Object(DSX_C_GDS_ACL_ITEM)	-	1 - 4	-
DSX_READ_SENSITIVE	Object(DSX_C_GDS_ACL_ITEM)	-	1 - 4	-
DSX_MODIFY_SENSITIVE	Object(DSX_C_GDS_ACL_ITEM)	-	1 - 4	-

The OM attributes of **DSX_C_GDS_ACL** are as follows:

- **DSX_MODIFY_PUBLIC**

This attribute specifies the user, or subtree of users, that can modify attributes classified as public attributes.

- **DSX_READ_STANDARD**

This attribute specifies the user, or subtree of users, that can read attributes classified as standard attributes.

- **DSX_MODIFY_STANDARD**

This attribute specifies the user, or subtree of users, that can modify attributes classified as standard attributes.

- **DSX_READ_SENSITIVE**

This attribute specifies the user, or subtree of users, that can read attributes classified as sensitive attributes.

- **DSX_MODIFY_SENSITIVE**

Specifies the user, or subtree of users, that can modify attributes classified as sensitive attributes.

DSX_C_GDS_ACL_ITEM

An instance of OM class **DSX_C_GDS_ACL_ITEM** is a component of a **DSX_C_GDS_ACL**. It specifies the user, or subtree of users, to whom an access right applies.

An instance of this OM class has the OM attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attributes listed in Table 80.

Table 80. OM Attributes of DSX_C_GDS_ACL_ITEM

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DSX_INTERPRETATION	Enum(DSX_Interpretation)	-	1	-
DSX_USER	Object(DSX_C_DS_DN)	-	1	-

The OM attributes of a **DSX_C_GDS_ACL_ITEM** are as follows:

- **DSX_INTERPRETATION**

This attribute specifies the scope of the access right. It can have one of the following values:

- **DSX_SINGLE_OBJECT**, Meaning that the access right is granted to the user specified in the **DSX_USER** OM attribute.

- **DSX_ROOT_OF_SUBTREE**, meaning that the access right is granted to all users in the subtree below the name specified in the **DSX_USER** OM attribute.

- **DSX_USER**

This attribute is the DN of the user, or subtree of users, to whom an access right applies.

DSX_C_GDS_CONTEXT

An instance of OM class **DSX_C_GDS_CONTEXT** comprises per-operation arguments that are accepted by most of the interface functions. GDSP supports additional service controls that are defined by the **DSX_C_GDS_CONTEXT** OM class.

An instance of this OM class has the OM attributes of its superclasses, *OM_C_OBJECT* and **DS_C_CONTEXT**, in addition to the OM attributes listed in Table 81.

Table 81. OM Attributes of a *DSX_C_GDS_CONTEXT*

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DSX_DUAFIRST	OM_S_BOOLEAN	-	1	OM_FALSE
DSX_DONT_STORE	OM_S_BOOLEAN	-	1	OM_TRUE
DSX_NORMAL_CLASS	OM_S_BOOLEAN	-	1	OM_FALSE
DSX_PRIV_CLASS	OM_S_BOOLEAN	-	1	OM_FALSE
DSX_RESIDENT_CLASS	OM_S_BOOLEAN	-	1	OM_FALSE
DSX_USEDSA	OM_S_BOOLEAN	-	1	OM_TRUE
DSX_DUA_CACHE	OM_S_BOOLEAN	-	1	OM_FALSE
DSX_PREFER_ADM_FUNCS	OM_S_BOOLEAN	-	1	OM_FALSE

The OM attributes of the **DSX_C_GDS_CONTEXT** OM class are as follows:

- **DSX_DUAFIRST**

This attribute defines whether the DUA cache or the DSA needs to be read first for query operations. The default value is **OM_FALSE**; that is, search DSA first, if not found then search DUA cache.

- **DSX_DONT_STORE**

This attribute specifies whether the information read from the DSAs by the query functions also needs to be stored in the DUA cache. When this service control is set to **OM_TRUE** (default value), nothing is stored in the DUA cache.

When this service control is set to **OM_FALSE**, the information read is stored in the DUA cache. The objects returned by **ds_list()** and **ds_compare()** are stored in the cache without their associated attribute information. The objects returned by **ds_read()** and **ds_search()** are stored in the cache with all the *cacheable* attributes; These are all public attributes that do not exceed 4 Kbytes in length.

This information is only cached when a list of requested attributes is supplied. If all attributes are requested, then nothing is stored in the cache.

The DUA cache categorizes the information stored into three different memory classes. The user specifies the category with the following service controls:

- **DSX_NORMAL_CLASS**

If this attribute is set to **OM_TRUE**, The entry in the DUA cache is assigned to the class of normal objects. If the number of entries in this class exceeds a maximum value, the entry that is not addressed for the longest period of time is removed from the DUA cache.

– **DSX_PRIV_CLASS**

If this attribute is set to **OM_TRUE**, the entry in the DUA cache is assigned to the class of privileged objects. Entries can be removed from the class in the same way as normal objects. By using this memory sparingly, the user can protect entries from deletion.

– **DSX_RESIDENT_CLASS**

If this attribute is set to **OM_TRUE**, the entry in the DUA cache is assigned to the class of resident objects. An entry in this memory class is never removed automatically, rather it can only be removed with **ds_remove_entry()**. The number of entries is limited; if this limit is exceeded, **ds_add_entry()** reports an error.

Only the service control of one memory class can be set. The **ds_add_entry()** function also evaluates these service control bits if the function is used on the DUA cache.

• **DSX_DUA_CACHE** and **DSX_USEDDSA**

These attributes define whether the entries in the DUA cache or in the DSA, or both, need to be used when providing the service. Depending on the values of these attributes, the following situations can arise:

– **DSX_DUA_CACHE=OM_TRUE** and **DSX_USEDDSA=OM_TRUE**

The **ds_add_entry()** and **ds_remove_entry()** functions report an error.

The query functions evaluate the service controls **DS_DONT_USE_COPY** and **DSX_DUAFIRST**. When **DS_DONT_USE_COPY** is **OM_FALSE**, then **DSX_DUAFIRST** determines whether the DUA cache or the DSA is read first. When **DS_DONT_USE_COPY** is **OM_TRUE**, information from the DSA only is read.

– **DSX_DUA_CACHE=OM_TRUE** and **DSX_USEDDSA=OM_FALSE**

The **ds_add_entry()** and **ds_remove_entry()** functions, and the query functions only go to the DUA cache.

– **DSX_DUA_CACHE=OM_FALSE** and **DSX_USEDDSA=OM_TRUE**

The **ds_add_entry()** and **ds_remove_entry()** functions and the query functions only go to the DSA.

– **DSX_DUA_CACHE=OM_FALSE** and **DSX_USEDDSA=OM_FALSE**

The **ds_add_entry()** and **ds_remove_entry()** functions and the query functions report an error.

All other functions always operate on the DSA currently connected.

• **DSX_PREFER_ADM_FUNCS**

GDS uses the three following optional attributes:

– **DSX_A_MASTER_KNOWLEDGE**, which contains the Distinguished Name of the DSA that holds the master copy of an entry.

– **DSX_A_ACL**, which is used for GDS access control.

– **DS_A_USER_PASSWORD** as an attribute of the **DS_O_DSA** object class, which is used by the GDS shadowing mechanism.

The **DSX_A_MASTER_KNOWLEDGE**, and **DSX_A_ACL** attributes are present in every GDS entry.

When an application requests all attributes, it can prevent any of these three optional attributes from being returned by setting this service control to **OM_FALSE**.

If GDS applications, for example, GDS administration, require these attributes, they are obtained by setting this service control to **OM_TRUE**.

Applications can assume that an object of OM class **DSX_C_GDS_CONTEXT**, created with default values of all its OM attributes, work with all the interface functions. The constant **DS_DEFAULT_CONTEXT** can be used as an argument to functions instead of creating an OM object with default values.

The default **DSX_C_GDS_CONTEXT** is defined in Table 82

<i>Table 82. Default DSX_C_GDS_CONTEXT</i>	
OM Attribute	Default Value
Common Arguments:	
DS_EXT	NULL
DS_OPERATION_PROGRESS	DS_OPERATION_NOT_STARTED
DS_ALIASED_RDNS	0
Service Controls:	
DS_CHAINING_PROHIB	OM_TRUE
DS_DONT_DEREFERENCE_ALIASES	OM_FALSE
DS_DONT_USE_COPY	OM_TRUE
DS_LOCAL_SCOPE	OM_FALSE
DS_PREFER_CHAINING	OM_FALSE
DS_PRIORITY	DS_MEDIUM
DS_SCOPE_OF_REFERRAL	—
DS_SIZE_LIMIT	-1
DS_TIME_LIMIT	-1
Local Controls:	
DS_ASYNCHRONOUS	OM_FALSE
DS_AUTOMATIC_CONTINUATION	OM_TRUE
Private Extensions:	
DSX_DUAFIRST	OM_FALSE
DSX_DONT_STORE	OM_TRUE
DSX_NORMAL_CLASS	OM_FALSE
DSX_PRIV_CLASS	OM_FALSE
DSX_RESIDENT_CLASS	OM_FALSE
DSX_USEDSA	OM_TRUE
DSX_DUA_CACHE	OM_FALSE
DSX_PREFER_ADM_FUNCS	OM_FALSE

DSX_C_GDS_SESSION

An instance of OM class **DSX_C_GDS_SESSION** identifies a particular link from an application program to a GDSP DUA. This additional OM class is necessary if the user either wants to specify a password as part of the user credentials, or wants to specify the GDSP directory identifier, or alternatively wants to specify both a password and the directory identifier. **DSX_C_GDS_SESSION** can be passed as an argument to **ds_bind()**.

An instance of this OM class has the OM attributes of its superclasses, *OM_C_OBJECT* and **DS_C_SESSION**, in addition to the OM attributes listed in Table 83.

Table 83. OM Attributes of DSX_C_GDS_SESSION

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DSX_PASSWORD	String(OM_S_OCTET_STRING)	-	0 or 1	-
DSX_DIR_ID	OM_S_INTEGER	-	1	1

The OM attributes of **DSX_C_GDS_SESSION** are as follows:

- **DSX_PASSWORD**

This attribute indicates the password for the user credentials.

- **DSX_DIR_ID**

This attribute contains an identifier for distinguishing between several configurations of the Directory Service within a GDS installation. The valid range is from 1 to 20.

Applications can assume that an object of OM class **DSX_C_GDS_SESSION**, created with default values of all its OM attributes, works with all the interface functions. Such a session can be created by passing the constant **DS_DEFAULT_SESSION** as an argument to **ds_bind()**, having already negotiated the GDSP package.

Table 84 defines **DS_DEFAULT_SESSION**.

Table 84. Default DSX_C_GDS_SESSION

OM Attribute	Default Value
DS_DSA_ADDRESS	Value obtained from the cache or absent.
DS_DSA_NAME	Value obtained from the cache or absent.
DS_FILE_DESCRIPTOR	DS_NO_VALID_FILE_DESCRIPTOR
DSX_DIR_ID	1

Note: The values of **DS_DSA_ADDRESS** and **DS_DSA_Name** are taken from the cache of Directory ID 1.

Chapter 16. Distributed Management Environment Support

The Distributed Management Environment (DME) Network Management Option (NMO) provides access to network management protocols. One of the protocols it supports is the CMIP protocol. CMIP uses names to identify and locate managed objects and management applications. GDS is used to provide this name to address resolution.

Note: z/OS DCE does not support DME. The DME naming information presented is intended to increase your understanding of DCE name structure and concepts.

DME has a requirement to support opaque address forms to cater to instances where CMIP is not running over pure OSI protocols. For this purpose, GDS contains some enhancements that are described in this chapter.

To support DME an additional directory object class, and an additional directory attribute were required. Additional OM classes or OM attributes were not necessary. Therefore, GDS supports DME without having to negotiate a specific XDS/DME package. An application must include **xdsdme.h** when using the new directory object class and attribute.

The concepts and notation used are first mentioned in “Introduction to OM Classes” on page 241. They are also fully explained in the following chapters:

- Chapter 17, “Information Syntaxes” on page 325
- Chapter 18, “XOM Service Interface” on page 331
- Chapter 19, “Object Management Package” on page 347

DME Attribute Types

This section presents the additional directory attribute type that DME uses. Each attribute type has an object identifier, which is the value of the OM attribute **DS_ATTRIBUTE_TYPE**. These object identifiers are represented in the interface by constants with the same name as the directory attribute, and they are prefixed by **DSX_A_** so that they can be easily identified.

This section contains two tables that are used to indicate the object identifier for the DME attribute type (see Table 85), and the values for the DME attribute type (see Table 86 on page 324), respectively. Following these two tables is a brief description of the attribute.

Table 85 shows the name and object identifier of the DME attribute type.

Note: This object identifier stems from the root **{iso(1) identified-organization(3) osf(22) dme(2) components(1) nmo(2) dmeNmoAttributeType(1)}**. Basic encoding rules state that the first two decimal numbers be combined according to the formula $x*40+y$ to form the first hexadecimal value. Thus 1.3.= 43 = \x2B\x18.

Table 85. Object Identifier for DME Attribute Type

Attribute Type	Object Identifier BER
	HEXADECIMAL
DSX_A_ALTERNATE_ADDRESS	\x2B\x18\x02\x01\x02\x01

Table 86 on page 324 shows the name of the attribute type, together with the OM value syntax used in the interface to represent values of that attribute type. The table also includes the range of lengths permitted for the string type, indicates whether the attribute can be multi-valued, and lists which matching rules are provided for the syntax.

Table 86. Representation of Values for DME Attribute Types

Attribute Type	OM Value Syntax	Value Length	Multi-Valued	Matching Rules
DSX_A_ALTERNATE_ADDRESS	String(OM_S_OCTET_STRING)	1-800	yes	E

The following is a description of the DME attribute:

- **DSX_A_ALTERNATE_ADDRESS**

This attribute is used by DME to store opaque address formats. In Table 86, it can be seen that the **AlternateAddress** attribute is stored internally by GDS as an octet string. The application expects the following syntax:

```
AlternateAddress ::= SEQUENCE {
    address      OCTET STRING,
    protocol     SET OF OBJECT IDENTIFIER }
```

DME Object Classes

The only additional DME object class is **DSX_O_DME_NMO_AGENT** (see Table 87). This object class has the same structure rules in the default schema as the application entity object class.

DSX_O_DME_NMO_AGENT is a subclass of **DS_O_APPLIC_ENTITY** (inherits the mandatory **DS_A_PRESENTATION_ADDRESS** and **DS_A_COMMON_NAME** attributes) and contains one attribute, **DSX_A_ALTERNATE_ADDRESS**.

Note: This object identifier stems from the root **{iso(1) identified-organization(3) osf(22) dme(2) components(1) nmo(2) dmeNmoObjectClass(2)}**. Basic encoding rules state that the first two decimal numbers be combined according to the formula $x*40+y$ to form the first hexadecimal value. Thus $1.3.= 43 = \backslashx2B\backslashx18$.

Table 87. Object Identifier for DME Object Class

Object Class	Object Identifier BER
	HEXADECIMAL
DSX_O_DME_NMO_AGENT	$\backslashx2B\backslashx18\backslashx02\backslashx01\backslashx02\backslashx02$

Chapter 17. Information Syntaxes

This chapter defines the syntaxes permitted for attribute values. The syntaxes are closely aligned with the types and type constructors of ASN.1. The **OM_value** data type specifies how a value of each syntax is represented in the C interface. (See “XOM Data Types” on page 331 for more information.)

Syntax Templates

The names of certain syntaxes are constructed from syntax templates. A syntax template is a lexical construct comprising a primary identifier followed by an asterisk enclosed in parentheses, as follows:

identifier (*)

A syntax template encompasses a group of related syntaxes. Any member of the group, without distinction, is indicated by the primary identifier (*identifier*) alone. A particular member is indicated by the template with the asterisk (*) replaced by one of a set of secondary identifiers associated with the template:

*identifier*₁ (*identifier*₂)

Syntaxes

A variety of syntaxes are defined. Most are functionally equivalent to ASN.1 types, as documented in the following sections in this chapter:

“Relationship to ASN.1 Simple Types” on page 327

“Relationship to ASN.1 Useful Types” on page 327

“Relationship to ASN.1 Character String Types” on page 328

“Relationship to ASN.1 Type Constructors” on page 328.

The following syntaxes are defined:

OM_S_BOOLEAN A value of this syntax is a Boolean; that is, can be **OM_TRUE** or **OM_FALSE**.

Enum(*) A value of any syntax encompassed by this syntax template is one of a set of values associated with the syntax. The only significant characteristic of the values is that they are distinct.

The group of syntaxes encompassed by this template is open ended. Zero or more members are added to the group by each package definition. The secondary identifiers that indicate the members are also assigned there.

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

OM_S_NULL The one value of this syntax is a valueless place holder.

Object(*) A value of any syntax encompassed by this syntax template is an object, which is any instance of a class associated with the syntax.

The group of syntaxes encompassed by this template is open ended. One member is added to the group by each class definition. The secondary identifier that indicates the member is the name of the class.

String(*) A value of any syntax encompassed by this syntax template is a string (as defined in “Strings” on page 326) whose form and meaning are associated with the syntax.

The group of syntaxes encompassed by this template is closed. One syntax is defined for each ASN.1 string type. The secondary identifier that indicates the member is, in general, the first word of the type’s name.

Strings

A *string* is an ordered sequence of zero or more bits, octets, or characters. A string is categorized as either a *bit string*, an *octet string*, or a *character string*, depending upon whether it contains bits, octets, or characters, respectively.

The value *length* of a string is the number of bits in a bit string, octets in an octet string, or characters in a character string. It is confined to the interval 0 to 2^{32} . Any constraints on the value length of a string are specified in the appropriate class definitions.

The syntaxes that form the String group are identified in Table 88, which shows the secondary identifier assigned to each such syntax. The identifiers in the first, second, and third columns indicate the syntaxes of bit, octet, and character strings, respectively. The String group comprises all syntaxes identified in the table.

Table 88. String Syntax Identifiers

Bit String Identifier	Octet String Identifier	Character String Identifier
OM_S_BIT_STRING	OM_S_ENCODING ¹	OM_S_GENERAL_STRING ²
	OM_S_OBJECT_IDENTIFIER_STRING ³	OM_S_GENERALISED_TIME_STRING ²
	OM_S_OCTET_STRING	OM_S_GRAPHIC_STRING ²
		OM_S_IA5_STRING ²
		OM_S_NUMERIC_STRING ²
		OM_S_OBJECT_DESCRIPTOR ²
		OM_S_PRINTABLE_STRING ²
		OM_S_TELETEX_STRING ²
		OM_S_UTC_TIME_STRING ²
		OM_S_VIDEOTEX_STRING ²
		OM_S_VISIBLE_STRING ²

Notes:

1. The octets are those the BER permits for the contents octets of the encoding of a value of any ASN.1 type.
 2. The characters are those permitted by ASN.1’s type of the same name. Values of these syntaxes are represented in their BER encoded form.
 3. The octets are those the BER permits for the contents octets of the encoding of a value of ASN.1’s object identifier type.
-

Representation of String Values

In the service interface, a string value is represented by a string data type. This is defined in “Strings” on page 326. The length of a string is the number of octets by which it is represented at the interface, in the range 0 to 2^{32} .

The length of a character string does not need to be equal to the number of characters it comprises because, for example, a single character can be represented using several octets.

Relationship to ASN.1 Simple Types

As shown in Table 89, for every ASN.1 simple type except Real, there is an OM syntax that is functionally equivalent to it. The simple types are listed in the first column of the table; the corresponding syntaxes in the second column.

Table 89. Syntax for ASN.1's Simple Types

Type	Syntax
Bit String	String(OM_S_BIT_STRING)
Boolean	OM_S_BOOLEAN
Integer	OM_S_NULL
Null	OM_S_NULL
Object Identifier	OM_S_OBJECT_IDENTIFIER_STRING
Octet String	OM_S_OCTET_STRING
Real	None

Relationship to ASN.1 Useful Types

As shown in Table 90, for every ASN.1 useful type, there is an OM syntax that is functionally equivalent to it. The useful types are listed in the first column of the table; the corresponding syntaxes in the second column.

Table 90. Syntax for ASN.1's Useful Types

Type	Syntax
External	Object (OM_C_EXTERNAL)
Generalized Time	OM_S_GENERALISED_TIME_STRING
Object Descriptor	OM_S_OBJECT_DESCRIPTOR_STRING
Universal Time	OM_S.UTC.TIME.STRING

Relationship to ASN.1 Character String Types

As shown in Table 91, for every ASN.1 character string type, there is an OM syntax that is functionally equivalent to it. The ASN.1 character string types are listed in the first column of the table; the corresponding syntax in the second column.

Table 91. Syntax for ASN.1's Character String Types

Type	Syntax
General String	OM_S_GENERAL_STRING
Graphic String	OM_S_GRAPHIC_STRING
IA5 String	OM_S_IA5_STRING
-	OM_S_LOCAL_STRING
Numeric String	OM_S_NUMERIC_STRING
Printable String	OM_S_PRINTABLE_STRING
Teletex String	OM_S_TELETEX_STRING
Videotex String	OM_S_VIDEOTEX_STRING
Visible String	OM_S_VISIBLE_STRING

Relationship to ASN.1 Type Constructors

As shown in Table 92, there are functionally equivalent OM syntaxes for some (but not all) ASN.1 type constructors. The constructors are listed in the first column of the table; the corresponding syntaxes are listed in the second column.

Table 92. Syntaxes for ASN.1's Type Constructors

Type Constructor	Syntax
Choice	OM_S_OBJECT
Enumerated	OM_S_ENUMERATION
Selection	none ¹
Sequence	OM_S_OBJECT
Sequence Of	OM_S_OBJECT
Set	OM_S_OBJECT
Set Of	OM_S_OBJECT
Tagged	none ²

Notes:

1. This type constructor, a purely specification-time phenomenon, has no corresponding syntax.
 2. This type constructor is used to distinguish the alternatives of a choice or the elements of a sequence or set. This function is performed by attribute types.
-

The effects of the principal type constructors can be achieved, in any of a variety of ways, using objects to group attributes or using attributes to group values. An OM application designer can (but need not) model these constructors as classes of the following kinds:

- Choice An attribute type can be defined for each alternative, with just one being permitted in an instance of the class.
- Sequence or Set An attribute type can be defined for each sequence or set element. If an element is optional, then the attribute has zero or one values.
- Sequence Of or Set Of A single, multi-valued attribute can be defined.

An ASN.1 definition of an Enumerated Type component of a structured type is generally mapped to an OM attribute with an OM syntax **OM_S_ENUMERATION** in this interface.

Chapter 18. XOM Service Interface

This chapter describes the following aspects of the XOM service interface:

- The conformance of the DCE X/Open OSI-Abstract-Data Manipulation implementation to the X/Open specification.
- The data types whose data values are the parameters and results of the functions that the service makes available to the client.
- An overview of the functions that the service makes available to the client. For a complete description of these functions, see the *z/OS DCE Application Development Reference*.
- The return codes that indicate the outcomes (in particular, the exceptions) that the functions can report.

See Chapter 7, “Example Application Programs” on page 159 for examples on using the XOM interface.

Note: In this chapter, the following notation [a, b) indicates a range from the integer a to integer b - 1. That is, the parenthesis indicates the value less one, whereas the square bracket indicates the value as it is.

Standards Conformance

The DCE XOM implementation conforms to the following specification:

X/Open CAE Specification, OSI-Abstract-Data Manipulation (XOM) (November 1991)

The following apply to the DCE XOM implementation:

- Multiple workspaces for XDS objects are supported.
- The OM package is supported.
- The **om_encode()** and **om_decode()** functions are not supported. The OM classes used by the DCE XDS/XOM API are not encodable.
- Translation to local character sets is provided.

XOM Data Types

The data types of the XOM service interface are defined in this section and in Table 93. These data types are repeated in the in *z/OS DCE Application Development Reference*.

Table 93 (Page 1 of 2). XOM Service Interface Data Types

Data Type	Description
OM_boolean	Type definition for a Boolean data value.
OM_descriptor	Type definition for describing an attribute type and value.
OM_enumeration	Type definition for an Enumerated data value.
OM_exclusions	Type definition for the <i>exclusions</i> parameter for om_get() .
OM_integer	Type definition for an Integer data value.
OM_modification	Type definition for the <i>modification</i> parameter for om_put() .

Table 93 (Page 2 of 2). XOM Service Interface Data Types

Data Type	Description
OM_object	Type definition for a handle to either a private or a public object.
OM_object_identifier	Type definition for an Object Identifier data value.
OM_private_object	Type definition for a handle to an object in an implementation-defined, or private, representation.
OM_public_object	Type definition for a defined representation of an object that can be directly interrogated by a programmer.
OM_return_code	Type definition for a value returned from all OM functions indicating either that the function succeeded or why it failed.
OM_string	Type definition for a data value of <i>String</i> syntaxes.
OM_syntax	Type definition for identifying a syntax type.
OM_type	Type definition for identifying an OM attribute type.
OM_type_list	Type definition for enumerating a sequence of OM attribute types.
OM_value	Type definition for representing any data value.
OM_value_position	Type definition for designating a particular location within a String data value.
OM_workspace	Type definition for identifying an application-specific API that implements OM, such as directory or message handling.

Some data types are defined in terms of the following intermediate data types, whose precise definitions in C are defined by the system:

- OM_sint** The positive and negative integers that can be represented in 16 bits
- OM_sint16** The positive and negative integers that can be represented in 16 bits
- OM_sint32** The positive and negative integers that can be represented in 32 bits
- OM_uint** The nonnegative integers that can be represented in 16 bits
- OM_uint16** The nonnegative integers that can be represented in 16 bits
- OM_uint32** The nonnegative integers which can be represented in 32 bits

Note: The **OM_sint** and **OM_uint** data types are defined by the range of integers they must accommodate. As typically declared in the C interface, they are defined by the range of integers permitted by the host machine's word size. The latter range, however, always encompasses the former.

The type definitions for these data types are as follows:

```
typedef int            OM_sint;
typedef short         OM_sint16;
typedef long int      OM_sint32;
typedef unsigned      OM_uint;
typedef unsigned short OM_uint16;
typedef long unsigned OM_uint32;
```

OM_boolean

The C declaration for an **OM_boolean** data value is as follows:

```
typedef OM_uint32 OM_boolean;
```

A data value of this data type is a Boolean, that is, either FALSE or TRUE.

FALSE (**OM_FALSE**) is indicated by 0 (zero). TRUE is indicated by any other integer, although the symbolic constant **OM_TRUE** refers to the integer 1 specifically.

OM_descriptor

The **OM_descriptor** data type is used to describe an attribute type and value. Its C declaration is as follows:

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

Note: Other components are encoded in the high bits of the syntax member.

A data value of this type is a descriptor, that embodies an attribute value. An array of descriptors can represent all the values of all the attributes of an object, and is the representation called

OM_public_object. A descriptor has the following components:

<i>type</i>	An OM_type data type. It identifies the data type of the attribute value.
<i>syntax</i>	An OM_syntax data type. It identifies the syntax of the attribute value. Components 3 to 7 (that is, the components <i>long_string</i> through <i>private</i> that follow) are encoded in the high-order bits of this structure member. Therefore, the syntax always needs to be masked with the constant OM_S_SYNTAX . For example: <pre>my_syntax = my_public_object[3].syntax & OM_S_SYNTAX; my_public_object[4].syntax = my_syntax + (my_public_object[4].syntax & OM_S_SYNTAX);</pre>
<i>long-string</i>	It is set only if the descriptor is a service-generated descriptor and the length of the value is greater than an implementation-defined limit. This component occupies bit 15 (0x8000) of the syntax and is represented by the constant OM_S_LONG_STRING .
<i>no-value</i>	It is set only if the descriptor is a service-generated descriptor and the value is not present because OM_EXCLUDE_VALUES or OM_EXCLUDE_MULTIPLES is set in om_get() . This component occupies bit 14 (0x4000) of the syntax and is represented by the constant OM_S_NO_VALUE .
<i>local-string</i>	Significant only if the syntax is one of the string syntaxes. It is set only if the string is represented in an implementation-defined local character set. The local character set may be more amenable for use as keyboard input or display output than the nonlocal character set, and can include specific treatment of line termination sequences. Certain interface functions can convert information in string syntaxes to or from the local representation, which may result in a loss of information.

This component occupies bit 13 (0x2000) of the syntax and is represented by the constant **OM_S_LOCAL_STRING**.

service-generated It is set only if the descriptor is a service-generated descriptor and the first descriptor of a public object, or the defined part of a private object. (See the *z/OS DCE Application Development Reference* for more information.)

This component occupies bit 12 (0x1000) of the syntax and is represented by the constant **OM_S_SERVICE_GENERATED**.

private It is set only if the descriptor in the service-generated public object contains a reference to the handle of a private subobject, or in the defined part of a private object.

Note: This applies only when the descriptor is a service-generated descriptor. The client need not set this bit in a client-generated descriptor that contains a reference to a private object.

This component occupies bit 11 (0x0800) of the syntax and is represented by the constant **OM_S_PRIVATE**.

value An **OM_value** data type. It identifies the attribute value.

OM_enumeration

The **OM_enumeration** data type is used to indicate an Enumerated data value. Its C declaration is as follows:

```
typedef OM_sint32 OM_enumeration;
```

A data value of this data type is an attribute value whose syntax is **OM_S_ENUMERATION**.

OM_exclusions

The **OM_exclusions** data type is used for the *exclusions* parameter of **om_get()**. Its C declaration is as follows:

```
typedef OM_uint OM_exclusions;
```

A data value of this data type is an unordered set of one or more values, all of which are distinct. Each value indicates an exclusion, as defined by **om_get()**, and is chosen from the following set:

- **OM_EXCLUDE_ALL_BUT_THESE_TYPES**
- **OM_EXCLUDE_MULTIPLES**
- **OM_EXCLUDE_ALL_BUT_THESE_VALUES**
- **OM_EXCLUDE_VALUES**
- **OM_EXCLUDE_SUBOBJECTS**
- **OM_EXCLUDE_DESCRIPTOR**

Alternatively, the single value **OM_NO_EXCLUSIONS** can be chosen; this selects the entire object.

Each value except **OM_NO_EXCLUSIONS** is represented by a distinct bit. The presence of the value is represented as 1; its absence is represented as 0 (zero). Thus, multiple exclusions are requested by ORing the values that indicate the individual exclusions.

OM_integer

The **OM_integer** data type is used to indicate an integer data value. Its C declaration is as follows:

```
typedef OM_sint32 OM_integer;
```

A data value of this data type is an attribute value whose syntax is **OM_S_INTEGER**.

OM_modification

The **OM_modification** data type is used for the *modification* parameter of **om_put()**. Its C declaration is as follows:

```
typedef OM_uint OM_modification;
```

A data value of this data type indicates a kind of modification, as defined by **om_put()**. It is chosen from the following set:

- **OM_INSERT_AT_BEGINNING**
- **OM_INSERT_AT_CERTAIN_POINT**
- **OM_INSERT_AT_END**
- **OM_REPLACE_ALL**
- **OM_REPLACE_CERTAIN_VALUES**

OM_object

The **OM_object** data type is used as a handle to either a private or a public object. Its C declaration is as follows:

```
typedef struct OM_descriptor_struct *OM_object;
```

A data value of this data type represents an object, which can be either public or private. It is an ordered sequence of one or more instances of the **OM_descriptor** data type. See the **OM_private_object** and **OM_public_object** data types for restrictions on that sequence (“OM_private_object” on page 337 and “OM_public_object” on page 337, respectively).

OM_object_identifier

The **OM_object_identifier** data type is used as an ASN.1 object identifier. Its C declaration is as follows:

```
typedef OM_string OM_object_identifier;
```

A data value of this data type contains an octet string that comprises the contents octets of the BER encoding of an ASN.1 object identifier.

C Declaration of Object Identifiers: Every application program that uses a class or other object identifier must explicitly import it into every compilation unit (C source module) that uses it. Each such class or object identifier name must be explicitly exported from just one compilation module. Most application programs find it convenient to export all the names they use from the same compilation unit. Exporting and importing is performed using the following two C macros:

- The importing macro makes the class or other object identifier constants available within a compilation unit.
 - **OM_IMPORT(class_name)**

- **OM_IMPORT**(*OID_name*)
- The exporting macro allocates memory for the constants that represent the class or another object identifier.
 - **OM_EXPORT**(*class_name*)
 - **OM_EXPORT**(*OID_name*)

Object identifiers are defined in the appropriate header files, with the definition identifier having the prefix **OMP_O_** followed by the variable name for the object identifier. The constant itself provides the hexadecimal value of the object identifier string.

Use of Object Identifiers in C: The following C macro initializes an OM descriptor with syntax **OM_S_OBJECT_IDENTIFIER_STRING**:

```
OM_OID_DESC(type, OID_name)
```

It sets the *type* component to that given, sets the *syntax* component to **OM_S_OBJECT_IDENTIFIER_STRING**, and sets the *value* component to the specified object identifier.

The following macro initializes a descriptor to mark the end of a client-allocated public object:

```
OM_NULL_DESCRIPTOR
```

For each class, there is a global variable of type **OM_STRING** with the same name; for example, the External class has a variable called **OM_C_EXTERNAL**.) This is also the case for other object identifiers; for example, the object identifier for BER rules has a variable called **OM_BER**. This global variable can be supplied as a parameter to functions when required.

This variable is valid only when it is exported by an **OM_EXPORT** macro and imported by an **OM_IMPORT** macro in the compilation units that use it. This variable cannot form part of a descriptor, but the value of its length and elements components can be used. The following code fragment provides examples of the use of the macros and constants:

```
/* Examples of the use of the macros and constants */

#include <xom.h>

OM_IMPORT(OM_C_ENCODING)
OM_IMPORT(OM_CANONICAL_BER)

/* The following sequence must appear in just one compilation
 * unit in place of the above:
 *
 * #include <xom.h>
 *
 * OM_EXPORT(OM_C_ENCODING)
 * OM_EXPORT(OM_CANONICAL_BER)
 */

main()
{
/* Use #1 - Define a public object of class Encoding
 *      (Note: xxxx is a Message Handling class which can be
 *      encoded)
 */
OM_descriptor my_public_object[] = {
    OM_OID_DESC(OM_CLASS, OM_C_ENCODING),
    OM_OID_DESC(OM_OBJECT_CLASS, MA_C_xxxx),
    { OM_OBJECT_ENCODING, OM_S_ENCODING, some_BER_value },
    OM_OID_DESC(OM_RULES, OM_CANONICAL_BER),
};
}
```

```

        OM_NULL_DESCRIPTOR
    };

/* Use #2 - Pass class Encoding as an argument to om_instance()
*/
return_code = om_instance(my_object, OM_C_ENCODING, &boolean_result);
}

```

OM_private_object

The **OM_private_object** data type is used as a handle to an object in an implementation-defined or private representation. Its C declaration is as follows:

```
typedef OM_object OM_private_object;
```

A data value of this data type is the designator or handle to a private object. It consists of a single descriptor whose *type* component is **OM_PRIVATE_OBJECT** and whose *syntax* and *value* components are unspecified.

Note: The descriptor's *syntax* and *value* components are essential to the service's proper operation with respect to the private object.

OM_public_object

The **OM_public_object** data type is used to define an object that can be directly accessed by a programmer. Its C declaration is as follows:

```
typedef OM_object OM_public_object;
```

A data value of this data type is a public object. It consists of one or more (usually more) descriptors; all but the last represent values of attributes of the object.

The descriptors for the values of a particular attribute with two or more values are adjacent to one another in the sequence. Their order is that of the values they represent. The order of the resulting groups of descriptors is unspecified.

Because the Class attribute specific to the Object class is represented among the descriptors, it must be represented before any other attributes. Regardless of whether or not the Class attribute is present, the *syntax* field of the first descriptor must have the **OM_S_SERVICE_GENERATED** bit set or cleared appropriately.

The last descriptor signals the end of the sequence of descriptors. The last descriptor's *type* component is **OM_NO_MORE_TYPES**, and its *syntax* component is **OM_S_NO_MORE_SYNTAXES**. The last descriptor's *value* component is unspecified.

OM_return_code

The **OM_return_code** data type is used for a value that is returned from all OM functions, indicating either that the function succeeded or the reason it failed. Its C declaration is as follows:

```
typedef OM_uint OM_return_code;
```

A data value of this data type is the integer in the range 0 to 2^{16} that indicates an outcome of an interface function. It is chosen from the set specified in "XOM Return Codes" on page 343.

Integers in the narrower range 0 to 2^{15} are used to indicate the return codes they define.

OM_string

The **OM_string** data type is used for a data value of String syntax. Its C declaration is as follows:

```
typedef OM_uint32 OM_string_length;
typedef struct {
    OM_string_length length;
    void *elements;
} OM_string;

#define OM_STRING(string)\
    { (OM_string_length)(sizeof(string)-1), (string) }
```

A data value of this data type is a string, that is, an instance of a String syntax. A string is specified either in terms of its length or whether or not it terminates with **NULL**. A string has the following components:

<i>length</i>	An OM_string_length data type. It is the number of octets by means of which the string is represented, or the OM_LENGTH_UNSPECIFIED value if the string terminates with null.
<i>elements</i>	The string's elements (that is, the octets that make up its value). The bits of a bit string are represented as a sequence of octets, as shown in Figure 49. The first octet stores the number of unused bits in the last octet. The bits in the bit string, commencing with the first bit and proceeding to the trailing bit, are placed in bits 7 to 0 of the second octet. These are followed by bits 7 to 0 of the third octet, then by bits 7 to 0 of each octet in turn, followed by as many bits as are required of the final octet, commencing with bit 7.

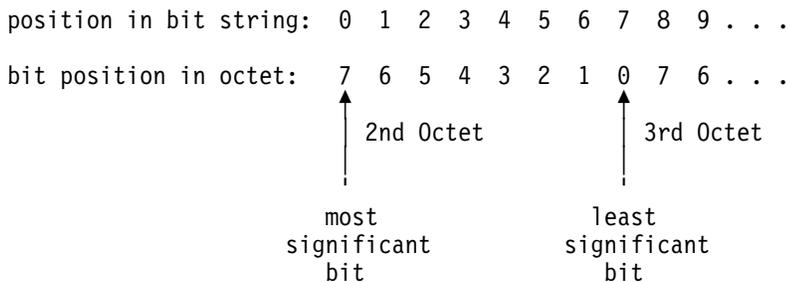


Figure 49. OM_String Elements

The service supplies a string value with a specified length. The client can supply a string value to the service in either form, either with a specified length or terminated with **NULL**.

The characters of a character string are represented as any sequence of octets permitted as the primitive contents octets of the BER encoding of an ASN.1 type value. The ASN.1 type defines the type of character string. A 0 value character follows the characters of the character string, but is not encompassed by the *length* component. Thus, depending upon the type of character string, the 0 value character can delimit the characters of the character string.

The **OM_STRING** macro is provided for creating a data value of this data type, given only the value of its *elements* component. The macro, however, applies to octet strings and character strings, but not to bit strings.

OM_syntax

The **OM_syntax** data type is used to identify a syntax type. Its C declaration is as follows:

```
typedef OM_uint16 OM_syntax;
```

A data value of this data type is an integer in the range 0 to 2^{10} that indicates an individual syntax or a set of syntaxes taken together.

The data value is chosen from among the following:

- **OM_S_BIT_STRING**
- **OM_S_BOOLEAN**
- **OM_S_ENCODING**
- **OM_S_ENUMERATION**
- **OM_S_GENERAL_STRING**
- **OM_S_GENERALISED_TIME_STRING**
- **OM_S_GRAPHIC_STRING**
- **OM_S_IA5_STRING**
- **OM_S_INTEGER**
- **OM_S_NULL**
- **OM_S_NUMERIC_STRING**
- **OM_S_OBJECT**
- **OM_S_OBJECT_DESCRIPTOR_STRING**
- **OM_S_OBJECT_IDENTIFIER_STRING**
- **OM_S_OCTET_STRING**
- **OM_S_PRINTABLE_STRING**
- **OM_S_TELETEX_STRING**
- **OM_S_VIDEOTEX_STRING**
- **OM_S_VISIBLE_STRING**
- **OM_S_UTC_TIME_STRING**

The integers in the range 2^9 to 2^{10} are reserved for vendor extensions. Wherever possible, the integers used are the same as the corresponding ASN.1 universal class number.

OM_type

The **OM_type** data type is used to identify an OM attribute type. Its C declaration is as follows:

```
typedef OM_uint16 OM_type;
```

A data value of this data type is an integer in the range 0 to 2^{16} that indicates a type in the context of a package. The following values in Table 94 on page 340 are assigned meanings by the respective data types:

Table 94. Assigning Meanings to Values

Value	Data Type
OM_NO_MORE_TYPES	OM_type_list
OM_PRIVATE_OBJECT	OM_private_object

OM_type_list

The **OM_type_list** data type is used to enumerate a sequence of OM attribute types. Its C declaration is as follows:

```
typedef OM_type *OM_type_list;
```

A data value of this data type is an ordered sequence of zero or more type numbers, each of which is an instance of the **OM_type** data type.

An additional data value, **OM_NO_MORE_TYPES**, follows and thus delimits the sequence. The C representation of the sequence is an array.

OM_value

The **OM_value** data type is used to represent any data value. Its C declaration is as follows:

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

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

Note: The first type definition (in particular, its **padding** component) aligns the **object** component with the *elements* component of the **string** component in the second type definition. This facilitates initialization in C.

The identifier **OM_value_union** is defined for reasons of compilation order. It is used in the definition of the **OM_descriptor** data type.

A data value of this data type is an attribute value. It has no components if the value's syntax is **OM_S_NO_MORE_SYNTAXES** or **OM_S_NO_VALUE**. Otherwise, it has one of the following components:

- string** The value if its syntax is a string syntax.
- boolean** The value if its syntax is **OM_S_BOOLEAN**.
- enumeration** The value if its syntax is **OM_S_ENUMERATION**.
- integer** The value if its syntax is **OM_S_INTEGER**.
- object** The value if its syntax is **OM_S_OBJECT**.

Note: A data value of this data type is only displayed as a component of a descriptor. Thus, it is always accompanied by indicators of the value's syntax. The latter indicator reveals which component is present.

OM_value_length

The **OM_value_length** data type is used to indicate the number of bits, octets, or characters in a string. Its C declaration is as follows:

```
typedef OM_uint32 OM_value_length;
```

A data value of this data type is an integer in the range 0 to 2^{32} that represents the number of bits in a bit string, octets in an octet string, or characters in a character string.

Note: This data type is not used in the definition of the interface. It is provided for use by client programmers for defining attribute constraints.

OM_value_position

The **OM_value_position** data type is used to indicate an attribute value's position within an attribute. Its C declaration is as follows:

```
typedef OM_uint32 OM_value_position;
```

A data value of this data type is an integer in the range 0 to $2^{32} - 1$ that indicates the position of a value within an attribute. The value **OM_ALL_VALUES** has the meaning assigned to it by **om_get()**.

OM_workspace

The **OM_workspace** data type is used to identify an application-specific API that implements OM, for example, directory or message handling. Its C declaration is as follows:

```
typedef void *OM_workspace;
```

A data value of this data type is the designator or handle for a workspace.

XOM Functions

This section provides an overview of the XOM service interface functions as listed in Table 95. For a full description of these functions, see the *z/OS DCE Application Development Reference*.

Table 95 (Page 1 of 2). XOM Service Interface Functions

Function	Description
om_copy()	Copies a private object.
om_copy_value()	Copies a string between private objects.
om_create()	Creates a private object.
om_decode()	This function is not supported by the DCE XOM interface; it returns with an OM_FUNCTION_DECLINED error.
om_delete()	Deletes a private or service-generated object.
om_encode()	This function is not supported by the DCE XOM interface; it returns with an OM_FUNCTION_DECLINED error.
om_get()	Gets copies of attribute values from a private object.
om_instance()	Tests an object's class.
om_put()	Puts attribute values into a private object.
om_read()	Reads a segment of a string in a private object.

Table 95 (Page 2 of 2). XOM Service Interface Functions

Function	Description
om_remove()	Removes attribute values from a private object.
om_write()	Writes a segment of a string into a private object.

The purpose and range of capabilities of the service interface functions can be summarized as follows:

om_copy()	This function creates an independent copy of an existing private object and all its subobjects. The copy is placed in the workspace of the original object, or in another workspace specified by the DCE client.
om_copy_value()	This function replaces an existing attribute value or inserts a new value in one private object with a copy of an existing attribute value found in another. Both values must be strings.
om_create()	<p>This function creates a new private object that is an instance of a particular class. The object can be initialized with the attribute values specified as initial in the class definition.</p> <p>The service does not permit the client to explicitly create instances of all classes, but rather only those indicated by a package's definition as having this property.</p>
om_delete()	This function deletes a service-generated public object, or makes a private object inaccessible.
om_get()	<p>This function creates a new public object that is an exact but independent copy of an existing private object. The client can request certain exclusions, each of which reduces the copy to a part of the original. The client can also request that values be converted from one syntax to another before they are returned.</p> <p>The copy can exclude: attributes of types other than those specified, values at positions other than those specified within an attribute, the values of multi-valued attributes, copies of (not handles for) subobjects, or all attribute values. Excluding all attributes values reveals only an attribute's presence.</p>
om_instance()	This function determines whether an object is an instance of a particular class. The client can determine an object's class simply by inspection. This function is useful because it reveals that an object is an instance of a particular class, even if the object is an instance of a subclass of that class.
om_put()	<p>This function places or replaces in one private object copies of the attribute values of another public or private object.</p> <p>The source values can be inserted before any existing destination values, before the value at a specified position in the destination attribute, or after any existing destination values. Alternatively, the source values can be substituted for any existing destination values or for the values at specified positions in the destination attribute.</p>
om_read()	This function reads a segment of a value of an attribute of a private object. The value must be a string. The value can first be converted from one syntax to another.
om_remove()	This function removes and discards particular values of an attribute of a private object. The attribute itself is removed if no values remain.
om_write()	This function writes a segment of an attribute value to a private object. The value must be a string. The segment can first be converted from one syntax to another. The written segment becomes the value's last segment since any elements beyond it are discarded.

XOM Return Codes

This section defines the return codes of the service interface, and thus the exceptions that can prevent the successful completion of an interface function. Table 96 identifies the abbreviated column headings that are used in Table 97; see Table 96 for the complete function names of the abbreviated column heads used in Table 97.

Table 97 lists the XOM return codes and the functions to which they apply. (The information in this table also appears in the ERRORS sections of the **XOM** function descriptions in the *z/OS DCE Application Development Reference*.) The first column of Table 97 lists the return codes. The other columns identify the return codes that apply to each function by means of an x.

Table 96. OM Functions and Their Corresponding Abbreviation

Function	Abbreviation
om_copy()	Cop
om_copy_value()	CoV
om_create()	Cre
om_decode()	Dec
om_delete()	Del
om_encode()	Enc
om_get()	Get
om_instance()	Ins
om_put()	Put
om_read()	Rea
om_remove()	Rem
om_write()	Wri

Table 97 (Page 1 of 2). XOM Service Interface Return Codes

Return Code	Cop	CoV	Cre	Dec	Del	Enc	Get	Ins	Put	Rea	Rem	Wri
OM_SUCCESS	x	x	x	x	x	x	x	x	x	x	x	x
OM_ENCODING_INVALID	-	-	-	x	-	-	-	-	-	-	-	-
OM_FUNCTION_DECLINED	-	x	x	-	-	x	-	-	x	-	x	x
OM_FUNCTION_INTERRUPTED	x	x	x	x	x	x	x	x	x	x	x	x
OM_MEMORY_INSUFFICIENT	x	x	x	x	x	x	x	x	x	x	x	x
OM_NETWORK_ERROR	x	x	x	x	x	x	x	x	x	x	x	x
OM_NO_SUCH_CLASS	x	-	x	x	-	-	-	x	x	-	-	-
OM_NO_SUCH_EXCLUSION	-	-	-	-	-	-	x	-	-	-	-	-
OM_NO_SUCH_MODIFICATION	-	-	-	-	-	-	-	-	x	-	-	-
OM_NO_SUCH_OBJECT	x	x	-	x	x	x	x	x	x	x	x	x
OM_NO_SUCH_RULES	-	-	-	x	-	x	-	-	-	-	-	-
OM_NO_SUCH_SYNTAX	-	-	-	-	x	-	-	x	x	-	-	x
OM_NO_SUCH_TYPE	-	x	-	-	x	-	x	-	x	x	x	x
OM_NO_SUCH_WORKSPACE	x	-	x	-	-	-	-	-	-	-	-	-
OM_NOT_AN_ENCODING	-	-	-	x	-	-	-	-	-	-	-	-
OM_NOT_CONCRETE	-	-	x	-	-	-	-	-	x	-	-	-
OM_NOT_PRESENT	-	x	-	-	-	-	-	-	x	x	-	x
OM_NOT_PRIVATE	x	x	-	x	-	x	x	-	x	x	x	x
OM_NOT_THE_SERVICES	-	-	-	-	x	-	-	x	-	-	-	-
OM_PERMANENT_ERROR	x	x	x	x	x	x	x	x	x	x	x	x
OM_POINTER_INVALID	x	x	x	x	x	x	x	x	x	x	x	x
OM_SYSTEM_ERROR	x	x	x	x	x	x	x	x	x	x	x	x
OM_TEMPORARY_ERROR	x	x	x	x	x	x	x	x	x	x	x	x
OM_TOO_MANY_VALUES	x	-	-	x	-	-	-	-	x	-	-	-
OM_VALUES_NOT_ADJACENT	-	-	-	-	-	-	-	-	x	-	-	-

Table 97 (Page 2 of 2). XOM Service Interface Return Codes

Return Code	Cop	CoV	Cre	Dec	Del	Enc	Get	Ins	Put	Rea	Rem	Wri
OM_WRONG_VALUE_LENGTH	-	x	-	x	-	-	-	-	x	-	-	x
OM_WRONG_VALUE_MAKEUP	-	-	-	x	-	-	-	-	x	-	-	x
OM_WRONG_VALUE_NUMBER	-	-	-	x	-	-	-	-	x	-	-	-
OM_WRONG_VALUE_SYNTAX	-	x	-	x	-	-	x	-	x	x	-	x
OM_WRONG_VALUE_TYPE	-	x	-	x	-	-	x	-	x	-	-	-

The return code values are as follows:

OM_SUCCESS	The function was completed successfully.
OM_ENCODING_INVALID	The octets that constitute the value of an encoding's Object Encoding attribute are invalid.
OM_FUNCTION_DECLINED	The function does not apply to the object to which it is addressed.
OM_FUNCTION_INTERRUPTED	The function is aborted by an external force; for example, a keystroke designated for this purpose, at a user interface.
OM_MEMORY_INSUFFICIENT	The service cannot allocate the main memory it needs to complete the function.
OM_NETWORK_ERROR	The service could not successfully employ the network upon which its implementation depends.
OM_NO_SUCH_CLASS	A purported class identifier is not defined.
OM_NO_SUCH_EXCLUSION	A purported exclusion identifier is not defined.
OM_NO_SUCH_MODIFICATION	A purported modification identifier is not defined.
OM_NO_SUCH_OBJECT	A purported object is nonexistent, or the purported handle is invalid.
OM_NO_SUCH_RULES	A purported rules identifier is not defined.
OM_NO_SUCH_SYNTAX	A purported syntax identifier is not defined.
OM_NO_SUCH_TYPE	A purported type identifier is not defined.
OM_NO_SUCH_WORKSPACE	A purported workspace is nonexistent.
OM_NOT_AN_ENCODING	An object is not an instance of the Encoding class.
OM_NOT_CONCRETE	A class is abstract, not concrete.
OM_NOT_PRESENT	An attribute value is absent, not present.
OM_NOT_PRIVATE	An object is public, not private.
OM_NOT_THE_SERVICES	An object is a client-generated object, rather than a service-generated or private object.
OM_PERMANENT_ERROR	The service encountered a permanent difficulty other than those indicated by other return codes.
OM_POINTER_INVALID	In the C interface, an invalid pointer is supplied as a function parameter, or as the receptacle for a function result.
OM_SYSTEM_ERROR	The service could not successfully employ the operating system upon which its implementation depends.
OM_TEMPORARY_ERROR	The service encountered a temporary difficulty other than those indicated by other return codes.
OM_TOO_MANY_VALUES	An implementation limit prevents a further attribute value from being added to an object. This limit is undefined.

OM_VALUES_NOT_ADJACENT	The descriptors for the values of a particular attribute are not adjacent.
OM_WRONG_VALUE_LENGTH	An attribute has, or would have, a value that violates the value length constraints in force.
OM_WRONG_VALUE_MAKEUP	An attribute has, or would have, a value that violates a constraint on the value's syntax.
OM_WRONG_VALUE_NUMBER	An attribute has, or would have, a value that violates the value number constraints in force.
OM_WRONG_VALUE_POSITION	The use defined for value position in the parameter or parameters of a function is invalid.
OM_WRONG_VALUE_SYNTAX	An attribute has, or would have, a value whose syntax is not permitted.
OM_WRONG_VALUE_TYPE	An object has, or would have, an attribute whose type is not permitted.

Chapter 19. Object Management Package

This chapter defines the Object Management Package (OMP). The object identifier (referred to as **OM**) assigned to the package, as defined in this book, is the object identifier specified in ASN.1 as **{joint-iso-ccitt(2) mhs-motis(6) group(6) white(1) api(2) om(4)}**.

Class Hierarchy

This section shows the hierarchical organization of the OM classes. Subclassification is indicated by indentation, and the names of abstract classes are in italics. Thus, for example, **OM_C_ENCODING** is an immediate subclass of *OM_C_OBJECT*, an abstract class. The names of classes to which **om_encode()** applies are in bold. (DCE XOM does not support the encoding of any **OM** classes.) The **om_create()** function applies to all concrete classes.

- *OM_C_OBJECT*
 - **OM_C_ENCODING**
 - **OM_C_EXTERNAL**

Class Definitions

The following subsections defines the OM classes.

OM_C_ENCODING

An instance of class **OM_C_ENCODING** is an object represented in a form suitable for transmission between workspaces, for transport via a network, or for storage in a file. Encoding can also be a suitable way of indicating to an intermediate service provider (for example, a directory, or message transfer system) an object that it does not recognize.

This class has the attributes of its superclass, *OM_C_OBJECT*, in addition to the specific attributes listed in Table 98.

Table 98. Attributes Specific to *OM_C_Encoding*

Attribute	Value Syntax	Value Length	Value Number	Value Initially
OM_OBJECT_CLASS	String(OM_S_OBJECT_IDENTIFIER_STRING)	-	1	-
OM_OBJECT_ENCODING	String ¹	-	1	-
OM_RULES	String(OM_S_OBJECT_IDENTIFIER_STRING)	-	1	OM_BER

Notes:

1. If the Rules attribute is **ber** or **canonical-ber**, the syntax of the present attribute must be **OM_S_ENCODING_STRING**.

OM_OBJECT_CLASS This attribute identifies the class of the object that the Object Encoding attribute encodes. The class must be concrete.

OM_OBJECT_ENCODING This attribute is the encoding itself.

OM_RULES

This attribute identifies the set of rules that are followed to produce the Object Encoding attribute. Among the defined values of this attribute are those represented as follows:

OM_BER This value is specified in ASN.1 as **{joint-iso-ccitt(2) asn1(1) basic-encoding(1)}**. This value indicates the BER.³

OM_CANONICAL_BER This value is specified in ASN.1 as **{joint-iso-ccitt(2) mhs-motis(6) group(6) white(1) api(2) om(4) canonical-ber(4)}**. This value indicates the canonical BER.⁴

OM_C_EXTERNAL

An instance of class **OM_C_EXTERNAL** is a data value and one or more information items that describe the data value and identify its data type. This class corresponds to ASN.1's External type, and thus the class and the attributes specific to it are more fully described indirectly in the specification of ASN.1.⁵

This class has the attributes of its superclass (*OM_C_OBJECT*) in addition to the OM attributes specific to this class that are listed in Table 99.

Table 99. Attributes Specific to *OM_C_External*

Attribute	Value Syntax	Value Length	Value Number	Value Initially
OM_ARBITRARY_ENCODING	String(OM_S_BIT_STRING)	-	0-1 ¹	-
OM_ASN1_ENCODING	String(OM_S_ENCODING_STRING)	-	0-1 ¹	-
OM_DATA_VALUE_DESCRIPTOR	String(OM_S_OBJECT_DESCRIPTOR_STRING)	-	0-1	-
OM_DIRECT_REFERENCE	String(OM_S_OBJECT_IDENTIFIER_STRING)	-	0-1	-
OM_INDIRECT_REFERENCE	String(OM_S_INTEGER)	-	0-1	-
OM_OCTET_ALIGNED_ENCODING	String(OM_S_OCTET_STRING)	-	0-1 ¹	-

Notes:

1. Only one of these three attributes is present.

OM_ARBITRARY_ENCODING This attribute is a representation of the data value as a bit string.

OM_ASN1_ENCODING The data value. This attribute can be present only if the data type is an ASN.1 type.

If this attribute value's syntax is an Object syntax, the data value's representation is that produced by **om_encode()** when its *Object* parameter is the attribute value and its *Rules* parameter is **ber**. Thus, the Object's class must be one to which **om_encode()** applies.

OM_DATA_VALUE_DESCRIPTOR This attribute contains a description of the data value.

³ See Clause 25.2 of Recommendation X.209, *Specification of Basic Encoding Rules for Abstract Syntax Notation 1 (ASN.1)*, CCITT Blue Book, Fascicle VIII.4, International Telecommunications Union, 1988. Also published by ISO as *ISO 8825*.

⁴ See Clause 8.7 of Recommendation X.509, *The Directory: Authentication Framework*, CCITT Blue Book, International Telecommunications Union, 1988. Also published by ISO as *ISO 9594-8*.

⁵ See Clause 34 of Recommendation X.208, *Specification of Abstract Syntax Notation 1 (ASN.1)*, CCITT Blue Book, Fascicle VIII.4, International Telecommunications Union, 1988. Also published by ISO as *ISO 8824*.

- OM_DIRECT_REFERENCE** This attribute contains a direct reference to the data type.
- OM_INDIRECT_REFERENCE** This attribute contains an indirect reference to the data type.
- OM_OCTET_ALIGNED_ENCODING** This attribute contains a representation of the data value as an octet string.

OM_C_OBJECT

The class **OM_C_OBJECT** represents information objects of any variety. This abstract class is distinguished by the fact that it has no superclass and that all other classes are its subclasses.

The attributes specific to this class are listed in Table 100.

Table 100. Attributes Specific to OM_C_Object

Attribute	Value Syntax	Value Length	Value Number	Value Initially
OM_CLASS	String(OM_S_OBJECT_IDENTIFIER_STRING)	-	1	-

OM_CLASS This attribute identifies the object's class.

Appendix A. Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Mail Station P300
522 South Road
Poughkeepsie, NY 12601-5400
U.S.A

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Trademarks

The following terms are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both.

IBM	AIX	BookManager
CICS	CICS/ESA	IBMLink
IMS	IMS/ESA	Library Reader
RACF	Resource Link	SecureWay
z/OS		

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, and service names may be trademarks or service marks of others.

Programming Interface Information

This *z/OS DCE Application Development Guide: Directory Services* documents intended Programming Interfaces that allow the customer to write programs to obtain services of z/OS DCE.

Glossary

This glossary defines technical terms and abbreviations used in z/OS DCE documentation. If you do not find the term you are looking for, refer to the index of the appropriate z/OS DCE manual or view the *IBM Glossary of Computing Terms*, located at:

<http://www.ibm.com/ibm/terminology>

This glossary includes terms and definitions from:

- *IBM Dictionary of Computing*, SC20-1699.
- *Information Technology—Portable Operating System Interface (POSIX)*, from the POSIX series of standards for applications and user interfaces to open systems, copyrighted by the Institute of Electrical and Electronics Engineers (IEEE).
- *American National Standard Dictionary for Information Systems*, ANSI X3.172-1990, copyright 1990 by the American National Standards Institute (ANSI). Copies may be purchased from the American National Standards Institute, 11 West 42nd Street, New York, New York 10036. Definitions are identified by the symbol (A) after the definition.
- *Information Technology Vocabulary*, developed by Subcommittee 1, Joint Technical Committee 1, of the International Organization for Standardization and the International Electrotechnical Commission (ISO/IEC JTC1.SC1).
- *CCITT Sixth Plenary Assembly Orange Book, Terms and Definitions* and working documents published by the International Telecommunication Union, Geneva, 1978.
- Open Software Foundation (OSF).

The following abbreviations indicate terms that are related to a particular DCE service:

CDS	Cell Directory Service
CICS/ESA®	Customer Information Control System/ESA
DTS	Distributed Time Service
GDS	Global Directory Service
IMS/ESA®	Information Management System/ESA
RPC	Remote Procedure Call
Security	Security Service
Threads	Threads Service
XDS	X/Open Directory Services

XOM X/Open OSI-Abstract-Data Manipulation

A

absolute time. A point on a time scale.

abstract class. GDS: An object management (OM) class of an OM object that forbids instances. An abstract class typically serves to describe the similarities between instances of two or more concrete classes.

abstract syntax notation one (ASN.1). A data representation scheme that enables complicated types to be defined and enables values of these types to be specified.

access control list (ACL). (1) GDS: Specifies the users with their access rights to an object. (2) Security: Data that controls access to a protected object. An ACL specifies the privilege attributes needed to access the object and the permissions that may be granted, to the protected object, to principals that possess such privilege attributes.

access point. GDS, XDS: The point at which an abstract service is obtained. A connection between a directory user agent (DUA) and a directory system agent (DSA).

access right. Synonym for *permission*.

accessible. Pertaining to an object whose client possesses a valid designator or handle.

account. Data in the Registry database that allows a principal to log in. An account is a registry object that relates to a principal.

ACF. Attribute configuration file.

ACL. Access control list.

active context handle. RPC: A context handle in RPC applications that the RPC has set to a non-null value and passed back to the calling program. The calling program supplies the active context handle in any future calls to procedures that share the same client context. See *client context* and *context handle*.

address. An unambiguous name, label, or number that identifies the location of a particular entity or service. See *presentation address*.

alias. Synonym for *alias name*.

alias entry. GDS: A directory entry, of object class *alias*, containing information used to provide an alternative name for an object.

alias name. (1) GDS: A name for a directory object that consists of one or more alias entries in the directory information tree (DIT). (2) Security: An optional alternate for a principal's primary name. Synonymous with *alias*. The alias shares the same UUID with the primary name.

aliased object. GDS: An object to which an alias entry refers.

anonymous user. A user who is not entered in the directory as an object and who logs in to the Global Directory Service without giving a name and password.

API. Application program interface.

application program interface (API). A functional interface supplied by the operating system or by a separately orderable licensed program that allows an application program written in a high-level language to use specific data or functions of the operating system or the licensed program.

application thread. A thread of execution created and managed by application code. See *client application thread*, *local application thread*, *RPC thread*, and *server application thread*.

architecture. (1) The organizational structure of a computer system, including the interrelationships among its hardware and software. (2) The logical structure and operating principles of a computer network. The operating principles of a network include those of services, functions, and protocols.

ASN.1. Abstract syntax notation one.

association (connection-oriented). A connection between a client and a server.

asynchronous. Without a regular time relationship; unexpected or unpredictable with respect to the running of program instructions.

asynchronous operation. An operation that occurs without a regular or predictable time relationship to a specified event; for example, the calling of an error diagnostic routine that may receive control at any time during the running of a computer program.

attribute. (1) RPC: An Interface Definition Language (IDL) or attribute configuration file (ACF) that conveys information about an interface, type, field, parameter, or operation. (2) DTS: A qualifier used with DTS commands. DTS has four attribute categories: characteristics, counters, identifiers, and status.

(3) XDS: Information of a particular type concerning an object and appearing in an entry that describes the object in the directory information base (DIB). It denotes the attribute's type and a sequence of one or more attribute values, each accompanied by an integer denoting the value's syntax.

attribute configuration file (ACF). RPC: An optional companion to an interface definition file that changes how the Interface Definition Language (IDL) compiler locally interprets the interface definition. See also *interface definition* and *Interface Definition Language*.

attribute syntax. GDS: A definition of the set of values that an attribute may assume. Attribute syntax includes the data type, in ASN.1, and usually one or more matching rules by which values may be compared.

attribute table. GDS: A recurring attribute of the directory schema with the description of the attribute types that are permitted.

attribute type. (1) XDS: The component of an attribute that indicates the type of information given by that attribute. Because it is an object identifier, it is unique among other attribute types. (2) XOM: Any of various categories into which the client dynamically groups values on the basis of their semantics. It is an integer unique only within the package.

attribute value. XDS, XOM: A particular instance of the type of information indicated by an attribute type.

attribute value assertion (AVA). GDS: An attribute type and attribute value pair. A relative distinguished name is comprised of one or more AVAs.

attribute value syntax. See *attribute syntax* and *syntax*.

authentication. In computer security, a method used to verify the identity of a principal.

authentication level. Synonym for *protection level*.

authentication protocol. A formal procedure for verifying a principal's network identity. Kerberos is an instance of a shared-secret authentication protocol.

Authentication Service. One of three services provided by the Security Service: it verifies principals according to a specified authentication protocol. The other Security services are the Privilege Service and the Registry Service.

authorization. (1) The determination of a principal's permissions with respect to a protected object. (2) The approval of a permission sought by a principal with respect to a protected object.

authorization protocol. A formal procedure for establishing the authorization of principals with respect to protected objects. Authorization protocols supported by the Security Service include DCE authorization and name-based authorization.

authorization service. RPC: An implementation of an authorization protocol.

AVA. Attribute value assertion.

B

Basic Encoding Rules (BER). A set of rules used to encode ASN.1 values as strings of octets.

BER. Basic Encoding Rules.

big endian. An attribute of data representation that reflects how multi-octet data is stored. In big endian representation, the lowest addressed octet of a multi-octet data item is the most significant. See *little endian*.

binary timestamp. An opaque 128-bit (16-octet) structure that represents a DTS time value.

binding. RPC: A relationship between a client and a server involved in a remote procedure call.

binding handle. RPC: A reference to a binding. See *binding information*.

binding information. RPC: Information about one or more potential bindings, including an RPC protocol sequence, a network address, an endpoint, at least one transfer syntax, and an RPC protocol version number. See *binding*. See also *endpoint*, *network address*, *RPC protocol*, *RPC protocol sequence*, and *transfer syntax*.

broadcast. A notification sent to all members within an arbitrary grouping such as nodes in a network or threads in a process. See also *signal*.

Browser. CDS: A Motif-based program that lets users view the contents and structure of a cell name space.

C

C interface. The interface that is defined at a level that depends on the variant of C standardized by ANSI.

cache. (1) CDS: The information that a CDS clerk stores locally to optimize name lookups. The cache contains attribute values resulting from previous lookups, as well as information about other clearinghouses and namespaces. (2) Security: Contains the credentials of a principal after the DCE login. (3) GDS: See *DUA cache*.

call thread. RPC: A thread created by an RPC server's runtime to run remote procedures. When engaged by a remote procedure call, a call thread temporarily forms part of the RPC thread of the call. See *application thread* and *RPC thread*.

cancel. (1) Threads: A mechanism by which a thread informs either itself or another thread to stop the thread as soon as possible. If a cancel arrives during an important operation, the canceled thread may continue until it can end the thread in a controlled manner. (2) RPC: A mechanism by which a client thread notifies a server thread (the canceled thread) to end the thread as soon as possible. See also *thread*.

CCITT. Consultative Committee on International Telegraphy and Telephone

CDS. Cell Directory Service.

CDS clerk. The software that provides an interface between client applications and CDS servers.

CDS control program (CDSCP). A command interface that CDS administrators use to control CDS servers and clerks and manage the name space and its contents. See also *manager*.

CDSCP. CDS control program.

cell. The basic unit of operation in the distributed computing environment. A cell is a group of users, systems, and resources that are grouped around a common purpose and that share common DCE services.

Cell Directory Service (CDS). A DCE component. A distributed replicated database service that stores names and attributes of resources located in a cell. CDS manages a database of information about the resources in a group of machines called a DCE cell.

cell-relative name. Synonym for *local name*.

chaining. GDS, XDS: A mode of interaction optionally used by a directory system agent (DSA) that cannot perform an operation itself. The DSA chains by calling the operation in another DSA and then relaying the outcome to the original requester.

child directory. CDS: A CDS directory that has a directory immediately above it is considered a child of that directory.

CICS. Customer Information Control System.

class. A category into which objects are placed on the basis of their purpose and internal structure.

clearinghouse. CDS: A collection of directory replicas on one CDS server. A clearinghouse takes the form of

a database file. It can exist only on a CDS server node; it cannot exist on a node running only CDS clerk software. Usually only one clearinghouse exists on a server node.

clerk. (1) DTS: A software component that synchronizes the clock for its client system by requesting time values from servers, calculating a new time from the values, and supplying the computed time to client applications. (2) CDS: A software component that receives CDS requests from a client application, ascertains an appropriate CDS server to process the requests, and returns the results of the requests to the client application.

client. A computer or process that accesses the data, services, or resources of another computer or process on the network. Contrast with *server*.

client application thread. RPC: A thread executing client application code that makes one or more remote procedure calls. See *application thread*, *local application thread*, *RPC thread*, and *server application thread*.

client binding information. Information about a calling client provided by the client runtime to the server runtime, including the address where the call originated, the RPC protocol used for the call, the requested object UUID, and client authentication information. See *binding information* and *server binding information*.

client context. RPC: The state within an RPC server generated by a set of remote procedures and maintained across a series of calls for a particular client. See *context handle*. See also *manager*.

client stub. RPC: The surrogate code for an RPC interface that is linked with and called by the client application code. In addition to general operations such as marshalling data, a client stub calls the RPC runtime to perform remote procedure calls and, optionally, to manage bindings. See *server stub*.

client/server model. A form of computing where one system, the client, requests something, and another system, the server, responds.

clock. The combined hardware interrupt timer and software register that maintains the system time.

code page. (1) A table showing codes assigned to character sets. (2) An assignment of graphic characters and control function meanings to all code points. (3) Arrays of code points representing characters that establish numeric order of characters. [OSF] (4) A particular assignment of hexadecimal identifiers to graphic elements. (5) Synonymous with code set. (6) See also *code point*, *extended character*.

collapse. CDS: To remove the contents of a directory from the display (close it) using the CDS Browser. To collapse an open directory, double-click on its icon. Double-clicking on a closed directory expands it. Contrast with *expand*.

compatible server. RPC: A server that offers the requested RPC interface and RPC object and that is accessible over a valid combination of network and transport protocols. It is supported by both the client and server RPC run times.

computed time. DTS: The resulting time after a DTS clock synchronization. The time value that the clerk or server process computes according to the values it receives from several servers.

concrete class. XOM: An OM class that permits instances.

condition variable. Threads: A synchronization object used in conjunction with a mutex. It allows a thread to suspend running until some condition is true.

connectionless protocol. RPC: A transport protocol such as UDP that does not require a connection to be established prior to data transfer. Contrast with *connection-oriented protocol*.

connection-oriented protocol. RPC: An RPC protocol that runs over a connection-based transport protocol. It is a connection-based, reliable, virtual-circuit transport protocol, such as TCP. Contrast with *connectionless protocol*.

Consultative Committee on International Telegraphy and Telephone (CCITT). A United Nations Specialized Standards group whose membership includes common carriers concerned with devising and proposing recommendations for international telecommunications representing alphabets, graphics, control information, and other fundamental information interchange issues.

context handle. RPC: A reference to state (client context) maintained across remote procedure calls by a server on behalf of a client. See *client context*.

continuation reference. GDS, XDS: A reference that describes how the performance of all or part of an operation can be continued at a different directory system agent (DSA) or DSAs. See also *referral*.

control access. CDS: An access right that grants users the ability to change the access control on a name and to perform other powerful management tasks, such as replicate a directory or move a clearinghouse.

convergence. CDS: The degree to which CDS attempts to keep all replicas of a directory consistent. Two factors control the persistence and speed at which CDS keeps directory replicas up to date: the setting of a

directory's **CDS_Convergence** attribute (high, medium, or low) and the background skulk time. By default, every directory inherits the convergence setting of its parent.

conversation key. Synonym for *session key*.

copy. GDS, XDS: Either a copy of an entry stored in other DSAs through bilateral agreement or a locally and dynamically stored copy of an entry resulting from a request (a cache copy).

creation timestamp (CTS). An attribute of all CDS clearinghouses, directories, soft links, child pointers, and object entries that contains a unique value reflecting the date and time the name was created. The timestamp consists of two parts; a time portion and a portion containing the system identifier of the node on which the name was created. These two parts guarantee uniqueness among timestamps generated on different nodes.

credentials. Security: A general term for privilege attribute data that has been certified by a trusted privilege certification authority.

CTS. Creation timestamp.

Customer Information Control System (CICS). An IBM licensed program that enables transactions entered at remote terminals to be processed concurrently by user-written application programs. It includes facilities for building, using, and maintaining databases.

D

daemon. (1) A long-lived process that runs unattended to perform continuous or periodic system-wide functions such as network control. Some daemons are triggered automatically to perform their task; others operate periodically. An example is the **cron** daemon, which periodically performs the tasks listed in the **crontab** file. Many standard dictionaries accept the spelling *demon*. (2) A DCE server process.

DAP. Directory access protocol.

Data Encryption Standard (DES). The National Institute of Standards and Technology (NIST) Data Encryption Standard, adopted by the U.S. government as Federal Information Processing Standard (FIPS) Publication 46, which allows only hardware implementations of the data encryption algorithm.

datagram. RPC: A network data packet that is independent of all other packets and does not guarantee delivery or sequentiality.

datagram protocol. RPC: A datagram-based transport protocol, such as User Datagram Protocol (UDP), that runs over a connectionless transport protocol.

DCE. Distributed Computing Environment.

DCEKERN. The address space that contains the DCE daemons.

decrypt. Security: To decipher data.

default DSA. GDS: The directory system agent (DSA) generally used when the user does not specify any particular DSA when connecting to the directory system.

default element. RPC: An optional profile element that contains a nil interface identifier and object UUID and that specifies a default profile. Each profile can contain only one default element. See *default profile*, *profile*, and *profile element*.

default profile. RPC: A backup profile referred to by the default element in another profile. The NSI import and lookup operations use the default profile, if present, whenever a search based on the current profile fails to find any useful binding information. See *default element* and *profile*.

DES. Data Encryption Standard.

descriptor. (1) XOM: The means by which the client and service exchange an attribute value and the integers that denote its representation, type, and syntax. (2) XDS: A defined data structure that is used to represent an OM attribute type and a single value.

descriptor list. GDS: An ordered sequence of descriptors that is used to represent several OM attribute types and values.

destructor. A user-supplied routine that is expected to finalize and then deallocate a per-thread context value.

DFS. Distributed File Service.

DIB. Directory information base.

directory. (1) A logical unit for storing entries under one name (the directory name) in a CDS namespace. Each physical instance of a directory is called a replica. (2) A collection of open systems that cooperates to hold a logical database of information about a set of objects in the real world.

directory access protocol (DAP). GDS: The protocol used by a DUA to access a DSA.

directory ID. Directory identifier.

directory identifier. GDS: An identifier for distinguishing several configurations of the directory service within an installation.

directory information base (DIB). GDS: The complete set of information to which the directory provides access, which includes all of the pieces of information that can be read or manipulated using the operations of the directory.

directory information tree (DIT). GDS: The directory information base (DIB) considered as a tree, whose vertices (other than the root) are the directory entries.

directory schema. GDS: The set of rules and constraints concerning directory information tree (DIT) structure, object class definitions, attribute types, and syntaxes that characterize the directory information base (DIB).

Directory Service. A DCE component. The Directory Service is a central repository for information about resources in a distributed system. See *Cell Directory Service* and *Global Directory Service*.

directory system. GDS: A system for managing a directory, consisting of one or more DSAs. Each DSA manages part of the DIB.

directory system agent (DSA). GDS: An open systems interconnection (OSI) application process that is part of the directory.

directory system protocol (DSP). GDS: The protocol used by a directory system agent (DSA) to access another DSA. The DSA runs in the GDS server machine and manages the GDS data base.

directory user agent (DUA). GDS: An open systems interconnection (OSI) application process that represents a user accessing the directory.

distinguished encoding. GDS, XDS: The restrictions to the Basic Encoding Rules designed to ensure a unique encoding of each ASN.1 value, defined in the X.500 Directory Standards (CCITT X.509).

distinguished name (DN). GDS: One of the names of an object, formed from the sequence of RDNs of its object entry and each of its superior entries.

distinguished value. GDS: An entry's attribute value that has been designated to appear in the RDN of the entry.

distributed computing. A type of computing that allows computers with different hardware and software to be combined on a network, to function as a single computer, and to share the task of processing application programs.

Distributed Computing Environment (DCE). A comprehensive, integrated set of services that supports the development, use, and maintenance of distributed applications. DCE is independent of the operating system and network; it provides interoperability and portability across heterogeneous platforms.

Distributed File Service (DFS). A DCE component. DFS joins the local file systems of several file server machines making the files equally available to all DFS client machines. DFS allows users to access and share files stored on a file server anywhere in the network, without having to consider the physical location of the file. Files are part of a single, global name space, so that a user can be found anywhere in the network by means of the same name.

distributed service. A DCE service that is used mainly by administrators to manage a distributed environment. These services include DTS, Security, and Directory.

Distributed Time Service (DTS). A DCE component. It provides a way to synchronize the times on different hosts in a distributed system.

DIT. Directory information tree.

DN. Distinguished name.

DNS. Domain Name System.

Domain Name System (DNS). A hierarchical scheme for giving meaningful names to hosts in a TCP/IP network.

domain name. A unique network name that is associated with a network's unique address.

DSA. Directory system agent.

DSP. Directory system protocol.

DTS. Distributed Time Service.

DTS entity. DTS: The server or clerk software on a system.

DUA. Directory user agent.

DUA cache. GDS: The part of the DUA that stores information to optimize name lookups. Each cache contains copies of recently accessed object entries as well as information about DSAs in the directory.

dynamic endpoint. RPC: An endpoint that is generated by the RPC runtime for an RPC server when the server registers its protocol sequences. It expires when the server stops running. See *endpoint* and *well-known endpoint*.

E

effective permissions. Security: The permissions granted to a principal as a result of a masking operation.

element. RPC: Any of the bits of a bit string, the octets of an octet string, or the octets by means of which the characters of a character string are represented.

encrypt. To systematically encode data so that it cannot be read without knowing the coding key.

encryption key. A value used to encrypt data so that only possessors of the encryption key can decipher it.

endian. An attribute of data representation that reflects how certain multi-octet data is stored in memory. See *big endian* and *little endian*.

endpoint. RPC: An address of a specific server instance on a host.

endpoint map. RPC: A database local to a node where local RPC servers register binding information associated with their interface identifiers and object identifiers. The endpoint map is maintained by the endpoint map service of the DCE daemon.

endpoint map service. RPC: A service that maintains a system's endpoint map for local RPC servers. When an RPC client makes a remote procedure call using a partially bound binding handle, the endpoint map service looks up the endpoint of a compatible local server. See *endpoint map*.

entity. (1) CDS: Any manageable element through the CDS namespace. Manageable elements include directories, object entries, servers, replicas, and clerks. The CDS control program (CDSCP) commands are based on directives targeted for specific entities. (2) DTS: See *DTS entity*.

entry. GDS, XDS: The part of the DIB that contains information relating to a single directory object. Each entry consists of directory attributes.

entry point vector (EPV). RPC: A list of addresses for the entry points of a set of remote procedures that starts the operations declared in an interface definition. The addresses are listed in the same order as the corresponding operation declarations.

ENV. environment variable

envelope. Security: Used to transport authentication data and conversation keys between the security server and principals.

environment variable (ENV). A variable included in the current software environment that is available to any called program that requests it.

EPV. Entry point vector.

exception. (1) An abnormal condition such as an I/O error encountered in processing a data set or a file. (2) One of five types of errors that can occur during a floating-point exception. These are valid operation, overflow, underflow, division by zero, and inexact results. [OSF] (3) Contrast with *interrupt*, *signal*.

expand. CDS: To display the contents of (open) a directory using the CDS Browser. A directory that is closed can be expanded by double-clicking on its icon. Double-clicking on an expanded directory collapses it. Contrast with *collapse*.

expiration age. RPC: The amount of time that a local copy of name service data from a NSI attribute remains unchanged before a request from an RPC application for the attribute requires its updating. See also *NSI attribute*.

export. (1) RPC: To place the server binding information associated with an RPC interface or a list of object UUIDs or both into an entry in a name service database. (2) To provide access information for an RPC interface. Contrast with *unexport*.

F

filter. An assertion about the presence or value of certain attributes of an entry to limit the scope of a search.

FIFO. first-in-first-out

first-in-first-out (FIFO). A queueing technique in which the next item to be retrieved is the item that has been in the queue the longest time.

foreign cell. A cell other than the one to which the local machine belongs. A foreign cell and its binding information are stored in either GDS or the Domain Name System (DNS). The act of contacting a foreign cell is called intercell. Contrast with *local cell*.

full name. CDS: The complete specification of a CDS name, including all parent directories in the path from the cell root to the entry being named.

full pointer. RPC: A pointer without the restrictions of a reference pointer.

fully bound binding handle. RPC: A server binding handle that contains a complete server address including an endpoint. Contrast with *partially bound binding handle*.

G

General-Use Programming Interface (GUPI). An interface, with few restrictions, for use in customer-written programs. The majority of programming interfaces are general-use programming interfaces, and are appropriate in a wide variety of application programs. A general-use programming interface requires the knowledge of the externals of the interface and perhaps the externals of related programming interfaces. Knowledge of the detailed design or implementation of the software product is not required.

GDA. Global Directory Agent.

GDS. Global Directory Service.

Global Directory Agent (GDA). A DCE component that makes it possible for the local CDS to access names in foreign cells. The GDA provides a connection to foreign cells through either the GDS or the Domain Name System (DNS).

Global Directory Service (GDS). A DCE component. A distributed replicated directory service that provides a global namespace that connects the local DCE cells into one worldwide hierarchy. DCE users can look up a name outside a local cell with GDS.

global name. A name that is universally meaningful and usable from anywhere in the DCE naming environment. The prefix /... indicates that a name is global.

group. (1) RPC: A name service entry that corresponds to one or more RPC servers that offer common RPC interfaces, RPC objects, or both. A group contains the names of the server entries, other groups, or both that are members of the group. See *NSI group attribute*. (2) Security: Data that associates a named set of principals that can be granted common access rights. See *subject identifier*.

group member. (1) RPC: A name service entry whose name occurs in the group. (2) Security: A principal whose name appears in a security group. See *group*.

H

handle. RPC: An opaque reference to information. See *binding handle*, *context handle*, *interface handle*, *name service handle*, and *thread handle*.

heterogeneous. Pertaining to a collection of dissimilar host computers such as those from different manufacturers. Contrast with *homogeneous*.

home cell. Synonym for *local cell*.

homogeneous. Pertaining to a collection of similar host computers such as those of one model or one manufacturer. Contrast with *heterogeneous*.

host ID. Synonym for *network address*.

I

IDL. Interface Definition Language.

IDL compiler. RPC: A compiler that processes an RPC interface definition and an optional attribute configuration file (ACF) to generate client and server stubs, and header files See *Interface Definition Language*.

immediate subclass. GDS: A subclass, of a class C, having no superclasses that are themselves subclasses of C.

immediate subordinate. GDS: In the directory information tree (DIT), an entry is an immediate subordinate of another if its distinguished name is formed by appending its relative distinguished name (RDN) to the distinguished name of the other entry.

immediate superclass. GDS: The superclass, of a class C, having no subclasses that are themselves superclasses of C.

immediate superior. GDS: In the directory information tree (DIT), an entry is the immediate superior of another if its distinguished name, followed by the relative distinguished name (RDN) of the other, forms the distinguished name of the other entry.

import. (1) RPC: To obtain binding information from a name service database about a server that offers a given RPC interface by calling the RPC NSI import operation. (2) RPC: To incorporate constant, type, and import declarations from one RPC interface definition into another RPC interface definition by means of the IDL import statement.

IMS. Information Management System.

inaccessible. Pertaining to an object for which the client does not possess a valid designator or handle.

inaccuracy. DTS: The bounded uncertainty of a clock value as compared to a standard reference.

information architecture. GDS: The representation of the information stored in object management (OM) objects and the hierarchical relationships between different classes of OM objects.

Information Management System (IMS). A database and data communication system capable of managing complex databases and networks in virtual storage.

instance. XOM: An object in the category represented by a class.

interface. RPC: A shared boundary between two or more functional units, defined by functional characteristics, signal characteristics, or other characteristics, as appropriate. The concept includes the specification of the connection of two devices having different functions. See *RPC interface*.

interface definition. RPC: A description of an RPC interface written in the DCE Interface Definition Language (IDL). See *RPC interface*.

Interface Definition Language (IDL). A high-level declarative language that provides syntax for interface definitions.

interface handle. RPC: A reference in code to an interface specification. See *binding handle* and *interface specification*.

interface identifier. RPC: A string containing the interface Universal Unique Identifier (UUID) and major and minor version numbers of a given RPC interface. See *RPC interface*.

interface specification. RPC: An opaque data structure that is generated by the DCE IDL compiler from an interface definition. It contains identifying and descriptive information about an RPC interface. See *interface definition*, *interface handle*, and *RPC interface*.

interface UUID. RPC: The Universal Unique Identifier (UUID) generated for an RPC interface definition using the UUID generator. See *interface definition* and *RPC interface*.

intermediate data type. Any of the basic data types in terms of which the other, substantive data types of the interface are defined.

International Organization for Standardization (ISO). An international body composed of the national standards organizations of 89 countries. ISO issues standards on a vast number of goods and services including networking software.

Internet address. The 32-bit address assigned to hosts in a TCP/IP network.

Internet Protocol (IP). In TCP/IP, a protocol that routes data from its source to its destination in an Internet environment. IP provides the interface from the higher level host-to-host protocols to the local network protocols. Addressing at this level is usually from host to host.

interval. DTS: The combination of a time value and the inaccuracy associated with it; the range of values represented by a combined time and inaccuracy notation. As an example, the interval 08:00.00I00:05:00 (eight o'clock, plus or minus five minutes) contains the time 07:57.00.

invoke ID. XDS: An integer used to distinguish one asynchronous (directory) operation from all other outstanding ones.

IP. Internet Protocol

ISO. International Organization for Standardization

J

junction. A specialized entry in the DCE namespace that contains binding information to enable communications between different DCE services.

K

Kerberos. The authentication protocol used to carry out DCE private key authentication. Kerberos was developed at the Massachusetts Institute of Technology.

key. A value used to encrypt and decrypt data.

key file. A file that contains encryption keys for noninteractive principals.

L

LAN. Local area network.

layer. In network architecture, a group of services, functions, and protocols that is complete from a conceptual point of view, that is one out of a set of hierarchically arranged groups, and that extends across all systems that conform to the network architecture.

LDAP. The Lightweight Directory Access Protocol API.

leaf entry. A directory entry that has no subordinates. It can be an alias entry or an object entry.

LFS. local file system

Lightweight Directory Access Protocol (LDAP). An interface that provides access through TCP/IP to directory services which accept the LDAP protocol.

little endian. An attribute of data representation that reflects how multi-octet data is stored. In little endian representation, the lowest addressed octet of a multi-octet data item is the least significant. See *big endian*.

local. (1) Pertaining to a device directly connected to a system without the use of a communication line.
(2) Pertaining to devices that have a direct, physical connection. Contrast with *remote*.

local application thread. RPC: An application thread that runs within the confines of one address space on a local system and passes control exclusively among local code segments. See *application thread*, *client application thread*, *RPC thread* and *server application thread*.

local area network (LAN). A network in which communication is limited to a moderate-sized geographical area (1 to 10 km) such as a single office building, warehouse, or campus, and which does not generally extend across public rights-of-way. A local network depends on a communication medium capable of moderate to high data rate (greater than 1Mbps), and normally operates with a consistently low error rate.

local cell. The cell to which the local machine belongs. Synonymous with *home cell*. Contrast with *foreign cell*.

local file system (LFS). An organized collection of data in the form of a root directory and its subdirectories and files. An LFS supports special features useful in a distributed environment: the ability to replicate data; to log file system data, enabling quick recovery after a crash; to simplify administration by dividing the file system into easily managed units called filesets; and to associate access control lists (ACLs) with files and directories. An LFS is located on a disk that is physically attached to a machine. In other file systems, a single disk partition contains only one file system. In DCE LFS an aggregate can contain multiple file systems (filesets). See also *access control list (ACL)*.

local name. A name that is meaningful and usable only within the cell where an entry exists. The local name is a shortened form of a global name. Local names begin with the prefix */.:* and do not contain a cell name. Synonymous with *cell-relative name*.

local server. DTS: A server that synchronizes with its peers and provides its clock value to other servers and clerks in the same network.

logical unit (LU). A host port through which a user gains access to the services of a network.

M

manager. RPC: A set of remote procedures that implement the operations of an RPC interface and that can be dedicated to a given type of object. See also *object* and *RPC interface*.

manager entry point vector. RPC: The runtime code

on the server side uses this entry point vector to dispatch incoming remote procedure calls. See *entry point vector* and *manager*.

marshalling. RPC: The process by which a stub converts local arguments into network data and packages the network data for transmission. Contrast with *unmarshalling*.

mask. (1) A pattern of characters used to control the retention or deletion of portions of another pattern of characters (2) Security: Used to establish maximum permissions that can then be applied to individual ACL entries. (3) GDS: The administration screen interface menus.

master entry. GDS: The original entry of an object. This is the entry in the directory system agent (DSA) that is specified in the master knowledge attribute of the entry.

master replica. CDS: The first instance of a specific directory in the namespace. After copies of the directory have been made, a different replica can be designated as the master, but only one master replica of a directory can exist at a time. CDS can create, update, and delete object entries and soft links in a master replica.

mutex. Mutual exclusion. A read/write lock that grants access to only a single thread at any one time. A mutex is often used to ensure that shared variables are always seen by other threads in a consistent way.

N

name. GDS, CDS: A construct that singles out a particular (directory) object from all other objects. A name must be unambiguous (denote only one object); however, it need not be unique (be the only name that unambiguously denotes the object).

name service. A central repository of named resources in a distributed system. In DCE, this is the same as Directory Service.

name service handle. RPC: An opaque reference to the context used by the series of next operations called during a specific name service interface (NSI) search or inquiry.

name service interface (NSI). RPC: A part of the application program interface (API) of the RPC run time. NSI routines access a name service, such as CDS, for RPC applications.

namespace. CDS: A complete set of CDS names that one or more CDS servers look up, manage, and share. These names can include directories, object entries, and soft links.

naming attribute. GDS: An attribute used to form the relative distinguished name (RDN) of an entry.

NCA. Network Computing Architecture.

NDR. Network Data Representation.

network. A collection of data processing products connected by communications lines for exchanging information between stations.

network address. An address that identifies a specific host on a network. Synonymous with *host ID*.

Network Computing Architecture (NCA). RPC: An architecture for distributing software applications across heterogeneous collections of networks, computers, and programming environments using UDP. NCA specifies part of the DCE Remote Procedure Call architecture.

network data. RPC: Data represented in a format defined by a transfer syntax. See also *transfer syntax*.

Network Data Representation (NDR). RPC: The transfer syntax defined by the Network Computing Architecture. See *transfer syntax*.

network protocol. A communications protocol from the Network Layer of the Open Systems Interconnection (OSI) network architecture, such as the Internet Protocol (IP).

node. (1) An endpoint of a link, or a junction common to two or more links in a network. Nodes can be preprocessors, controllers, or workstations, and they can vary in routing and other functional capabilities. (2) In network topology, the point at an end of a branch. It is usually a physical machine.

nonspecific subordinate reference. GDS: A knowledge reference that holds information about the directory system agent (DSA) that holds one or more unspecified subordinate entries.

NSI. Name service interface.

NSI attribute. RPC: An RPC-defined attribute of a name service entry used by the RPC name service interface. A name service interface (NSI) attribute stores one of the following: binding information, object Universal Unique Identifiers (UUIDs), a group, or a profile. See *NSI binding attribute*, *NSI group attribute*, *NSI object attribute*, and *NSI profile attribute*.

NSI binding attribute. RPC: An RPC-defined attribute (NSI attribute) of a name service entry; the binding attribute stores binding information for one or more interface identifiers offered by an RPC server and identifies the entry as an RPC server entry. See

binding information and *NSI object attribute*. See also *server entry*.

NSI group attribute. RPC: An RPC-defined attribute (NSI attribute) of a name service entry that stores the entry names of the members of an RPC group and identifies the entry as an RPC group. See *group*.

NSI object attribute. RPC: An RPC-defined attribute (NSI attribute) of a name service entry that stores the object UUIDs of a set of RPC objects. See *object*.

NSI profile attribute. RPC: An RPC-defined attribute (NSI attribute) of a name service entry that stores a collection of RPC profile elements and identifies the entry as an RPC profile. See *profile*.

NULL. In the C language, a pointer that does not point to a data object.

O

object. (1) A data structure that implements some feature and has an associated set of operations. (2) RPC: For RPC applications, anything that an RPC server defines and identifies to its clients using an object Universal Unique Identifier (UUID). An RPC object is often a physical computing resource such as a database, directory, device, or processor. Alternatively, an RPC object can be an abstraction that is meaningful to an application, such as a service or the location of a server. See *object UUID*. (3) XDS: Anything in the world of telecommunications and information processing that can be named and for which the directory information base (DIB) contains information. (4) XOM: Any of the complex information objects created, examined, changed, or destroyed by means of the interface.

object class. GDS, CDS: An identified family of objects that share certain characteristics. An object class can be specific to one application or shared among a group of applications. An application interprets and uses an entry's class-specific attributes based on the class of the object that the entry describes.

object class table (OCT). A recurring attribute of the directory schema with the description of the object classes permitted.

object entry. CDS: The name of a resource (such as a node, disk, or application) and its associated attributes, as stored by CDS. CDS administrators, client application users, or the client applications themselves can give a resource an object name. CDS supplies some attribute information (such as a creation timestamp) to become part of the object, and the client application may supply more information for CDS to store as other attributes. See *entry*.

object identifier (OID). A value (distinguishable from all other such values) that is associated with an information object. It is formally defined in the CCITT X.208 standard.

object management (OM). The creation, examination, change, and deletion of potentially complex information objects.

object name. CDS: A name for a network resource.

object UUID. RPC: The Universal Unique Identifier (UUID) that identifies a particular RPC object. A server specifies a distinct object UUID for each of its RPC objects. To access a particular RPC object, a client uses the object UUID to find the server that offers the object. See *object*.

OCT. Object class table.

octet. A byte that consists of eight bits.

OID. Object identifier.

OM. Object management.

OM attribute. XOM: An object management (OM) attribute consists of one or more values of a particular type (and therefore syntax).

OM class. XOM: A static grouping of object management (OM) objects, within a specification, based on both their semantics and their form.

opaque. A datum or data type whose contents are not visible to the application routines that use it.

Open Software Foundation (OSF). A nonprofit research and development organization set up to encourage the development of solutions that allow computers from different vendors to work together in a true open-system computing environment.

open system. A system whose characteristics comply with standards made available throughout the industry and that can be connected to other systems complying with the same standards.

open systems interconnection (OSI). The interconnection of open systems in accordance with standards of the International Organization for Standardization (ISO) for the exchange of information.

operation. (1) GDS: Processing performed within the directory to provide a service, such as a read operation. (2) RPC: The task performed by a routine or procedure that is requested by a remote procedure call.

organization. (1) The third field of a subject identifier. (2) Security: Data that associates a named set of users

who can be granted common access rights that are usually associated with administrative policy.

OSF. Open Software Foundation.

OSI. Open systems interconnection

P

PAC. Privilege attribute certificate.

package. XOM: A specified group of related object management (OM) classes, denoted by an object identifier.

package closure. XOM: The set of classes that need to be supported to create all possible instances of all classes defined in the package.

packet. (1) In data communication, a sequence of binary digits, including data and control signals, that is transmitted and switched as a composite whole. [1] The data, call control signals, and error control information are arranged in a specific format. (2) See *call-accepted packet*, *call-connected packet*, *call-request packet*. See *clear-confirmation packet*, *clear-indication packet*, *clear-request packet*. See *data packet*, *incoming-call packet*.

parent directory. CDS: Any directory that has one or more levels of directories beneath it in a cell name space. A directory is the parent of any directory immediately beneath it in the hierarchy.

partially bound binding handle. RPC: A server binding handle that contains an incomplete server address lacking an endpoint. Contrast with *fully bound binding handle*.

Partitioned data set (PDS). A data set in direct access storage that is divided into partitions, called members, each of which can contain a program, part of a program, or data.

password. A secret string of characters shared between a computer system and a user. The user must specify the character string to gain access to the system.

PDS. Partitioned data set

permission. (1) The modes of access to a protected object. The number and meaning of permissions with respect to an object are defined by the access control list (ACL) Manager of the object. (2) GDS: One of five groups that assigns modes of access to users: MODIFY PUBLIC, READ STANDARD, MODIFY STANDARD, READ SENSITIVE, or MODIFY SENSITIVE. Synonymous with *access right*. See also *access control list*.

person. See *principal*.

pipe. (1) RPC: A mechanism for passing large amounts of data in a remote procedure call. (2) The data structure that represents this mechanism.

plaintext. The input to an encryption function or the output of a decryption function. Encryption transforms plaintext to ciphertext and decryption transforms ciphertext into plaintext.

platform. The operating system environment in which a program runs.

port. (1) Part of an Internet Protocol (IP) address specifying an endpoint. (2) To make the programming changes necessary to allow a program that runs on one type of computer to run on another type of computer.

position (within a string). XOM: The ordinal position of one element of a string relative to another.

position (within an attribute). XOM: The ordinal position of one value relative to another.

predicate. A Boolean logic term denoting a logical expression that determines the state of some variables. For example, a predicate can be an expression stating that variable A must have the value 3. The control expression used in conjunction with condition variables is based on a predicate. A condition variable can be used to wait for some predicate to become true, for example, to wait for something to be in a queue.

presentation address. An unambiguous name that is used to identify a set of presentation service access points. Loosely, it is the network address of an open systems interconnection (OSI) service.

presentation service access point (PSAP). Address of an open systems interconnection (OSI) communications partner. It addresses an application in a computer.

primary name. The string name of an object to which any aliases for that object refer. The DCE refers to objects by their primary names, although DCE users may refer to them by their aliases.

principal. Security: An entity that can communicate securely with another entity. In the DCE, principals are represented as entries in the Registry database and include users, servers, computers, and authentication surrogates.

privacy. RPC: A protection level that encrypts RPC argument values. in secure RPC communications.

private key. See *secret key*.

private object. (1) XDS: An OM object created in a work space using the object management functions. Contrast with public object. (2) XOM: An object that is represented in an unspecified fashion.

privilege attribute. Security: An attribute of a principal that may be associated with a set of permissions. DCE privilege attributes are identity-based and include the principal's name, group memberships, and local cell.

privilege attribute certificate (PAC). Security: Data describing a principal's privilege attributes that has been certified by an authority. In the DCE, the Privilege Service is the certifying authority; it seals the privilege attribute data in a ticket. The authorization protocol, DCE Authorization, determines the permissions granted to principals by comparing the privilege attributes in PACs with entries in an access control list.

privilege service. Security: One of three services provided by the Security Service; the Privilege Service certifies a principal's privileges. The other services are the Registry Service and the Authentication Service.

privilege ticket. Security: A ticket that contains the same information as a simple ticket, and also includes a privilege attribute certificate. See *service ticket*, *simple ticket*, and *ticket-granting ticket*.

procedure declaration. RPC: The syntax for an operation, including its name, the data type of the value it returns (if any), and the number, order, and data types of its parameters (if any).

product-sensitive programming interface (PSPI). (1) A special interface that is intended only to be used for a specialized task such as diagnosis, modification, monitoring, repairing, tailoring, or tuning. (2) A special interface that is dependent on or requires the customer to understand significant aspects of the detailed design and implementation of the IBM software product.

profile. RPC: An entry in a name service database that contains a collection of elements from which name service interface (NSI) search operations construct search paths for the database. Each search path is composed of one or more elements that refer to name service entries corresponding to a given RPC interface and, optionally, to an object. See *NSI profile attribute* and *profile element*.

profile element. RPC: A record in an RPC profile that maps an RPC interface identifier to a profile member (a server entry, group, or profile in a name service database). See *profile*. See also *group*, *interface identifier* and *server entry*.

profile member. RPC: A name service entry whose name occupies the member field of an element of the profile. See *profile*.

programming interface. The supported method through which customer programs request software services. The programming interface consists of a set of callable services provided with the product.

proprietary. Pertaining to the holding of the exclusive legal rights in making, using, or marketing a product.

protection level. The degree to which secure network communications are protected. Synonymous with *authentication level*.

protocol. A set of semantic and syntactic rules that determines the behavior of functional units in achieving communication.

protocol sequence. Synonym for *RPC protocol sequence*.

protocol sequence vector. RPC: A data structure that contains an array-size count and an array of pointers to RPC protocol-sequence strings. See *RPC protocol sequence*.

PSAP. Presentation service access point.

public object. (1) XOM: An object that is represented by a data structure whose format is part of the service's specification. (2) XDS: A descriptor list that contains all the OM attributes of an OM object.

purported name. A construct that is syntactically a name, but has not yet been shown to be a valid name.

R

RACF. Resource Access Control Facility.

RDN. Relative distinguished name.

read access. CDS: An access right that grants the ability to view data.

read-only replica. (1) CDS: A copy of a CDS directory in which applications cannot make changes. Although applications can look up information (read) from it, they cannot create, change, or delete entries in a read-only replica. Read-only replicas become consistent with other, changeable replicas of the same directory during skulks and routine propagation of updates. (2) Security: A replicated Registry server.

realm. Security: A cell, considered exclusively from the point of view of Security; this term is used in Kerberos specifications. The term cell designates the basic unit of DCE configuration and administration and incorporates the notion of a realm.

recurring attribute. An attribute with several attribute values.

reference monitor. Code that controls access to an object. In the DCE, servers control access to the objects they maintain; for a given object, the ACL manager associated with that object makes authorization decisions concerning the object.

reference pointer. RPC: A non-null pointer whose value is invariant during a remote procedure call and cannot point at aliased storage.

referral. GDS: An outcome that can be returned by a DSA that cannot perform an operation itself. The referral identifies one or more other DSAs more able to perform the operation.

register. (1) RPC: To list an RPC interface with the RPC runtime. (2) To place server-addressing information into the local endpoint map. (3) To insert authorization and authentication information into binding information. See *endpoint map* and *RPC interface*.

Registry database. Security: A database of security information about principals, groups, organizations, accounts, and security policies.

Registry Service. Security: One of three services provided by the Security Service; the Registry Service manages information about principals, accounts, and security policies. The other services are the Privilege Service and the Authentication Service.

relative distinguished name (RDN). GDS, XDS: A set of Attribute Value Assertions (AVAs).

relative time. A discrete time interval that is usually added to or subtracted from an absolute time. See *absolute time*.

remote. Pertaining to a device, file or system that is accessed by your system through a communications line. Contrast with *local*.

remote procedure. RPC: An application procedure located in a separate address space from calling code. See *remote procedure call*.

remote procedure call. RPC: A client request to a service provider located anywhere in the network.

Remote Procedure Call (RPC). A DCE component. It allows requests from a client program to access a procedure located anywhere in the network.

replica. CDS: A directory in the CDS namespace. The first instance of a directory in the name space is the master replica. See *master replica* and *read-only replica*.

replication. The making of a shadow of a database to be used by another node. Replication can improve availability and load-sharing.

request. A command sent to a server over a connection.

resource. Items such as printers, plotters, data storage, or computer services. Each has a unique identifier associated with it for naming purposes.

Resource Access Control Facility (RACF). An IBM licensed program, that provides for access control by identifying and verifying the users to the system, authorizing access to protected resources, and logging the detected unauthorized access to protected resources.

return value. A function result that is returned in addition to the values of any output or input/output arguments.

ROM. Read-only memory.

RPC. Remote Procedure Call.

RPC control program (RPCCP). An interactive administrative facility for managing name service entries and endpoint maps for RPC applications.

RPCCP. RPC control program

RPC interface. A logical group of operations, data types, and constant declarations that serves as a network contract for a client to request a procedure in a server. See also *interface definition* and *operation*.

RPC protocol. An RPC-specific communications protocol that supports the semantics of the DCE RPC API and runs over either connectionless or connection-oriented communications protocols.

RPC protocol sequence. A valid combination of communications protocols represented by a character string. Each RPC protocol sequence typically includes three protocols: a network protocol, a transport protocol, and an RPC protocol that works with the network and transport protocols. See *network protocol*, *RPC protocol*, and *transfer protocol*. Synonymous with *protocol sequence*.

RPC runtime. A set of operations that manages communications, provides access to the name service database, and performs other tasks, such as managing servers and accessing security information, for RPC applications. See *RPC runtime library*.

RPC runtime library. A group of routines of the RPC runtime that support the RPC applications on a system. The runtime library provides a public interface to application programmers, the application programming interface (API), and a private interface to stubs, the stub programming interface (SPI). See *RPC runtime*.

RPC thread. A logical thread within which a remote procedure call is executed. See *thread*.

S

schema. See *directory schema*.

secret key. Security: A long-lived encryption key shared between a principal and the Authentication Service.

Security Service. A DCE component that provides trustworthy identification of users, secure communications, and controlled access to resources in a distributed system.

segment. One or more contiguous elements of a string.

server. (1) On a network, the computer that contains programs, data, or provides the facilities that other computers on the network can access. (2) The party that receives remote procedure calls. Contrast with *client*.

server addressing information. RPC: An RPC protocol sequence, network address, and endpoint that represent one way to access an RPC server over a network; a part of server binding information. See *network address*. See also *binding information*, *endpoint*, and *RPC protocol sequence*.

server application thread. RPC: A thread running the server application code that initializes the server and listens for incoming calls. See *application thread*, *client application thread*, *local application thread*, and *RPC thread*.

server binding information. RPC: Binding information for a particular RPC server. See *binding information* and *client binding information*.

server entry. RPC: A name service entry that stores the binding information associated with the RPC interfaces of a particular RPC server and object Universal Unique Identifiers (UUIDs) for any objects offered by the server. See also *binding information*, *NSI binding attribute*, *NSI object attribute*, *object* and *RPC interface*.

server instance. RPC: A server running in a specific address space. See *server*.

server stub. RPC: The surrogate calling code for an RPC interface that is linked with server application code containing one or more sets of remote procedures (managers) that implement the interface. See *client stub*. See also *manager*.

service. In network architecture, the capabilities that the layers closer to the physical media provide to the layers closer to the end user.

service controls. GDS, XDS: A group of parameters, applied to all directory operations, that direct or constrain the provision of the service.

service ticket. Security: A ticket for a specified service other than the ticket-granting service. See *privilege ticket*, *simple ticket*, and *ticket-granting ticket*.

session. GDS: A sequence of directory operations requested by a particular user of a particular directory user agent (DUA) using the same session object management (OM) object.

session key. Security: A short-lived encryption key provided by the Authentication Service to two principals for the purpose of ensuring secure communications between them. Synonymous with *conversation key*.

shadow entry. GDS: A copy entry of an object. This is an entry of an object in a directory system agent (DSA) other than the master DSA.

SID. Subject identifier.

signal. Threads: To wake only one thread waiting on a condition variable. See *broadcast*.

signed. Security: Pertaining to information that is appended to an enciphered summary of the information. This information is used to ensure the integrity of the data, the authenticity of the originator, and the unambiguous relationship between the originator and the data.

simple name. CDS: One element in a CDS full name. Simple names are separated by slashes in the full name.

simple ticket. Security: A ticket that contains the principal's identity, a session key, a timestamp and other information, sealed using the target's secret key. See *privilege ticket*, *service ticket*, and *ticket-granting ticket*.

socket. A unique host identifier created by the concatenation of a port identifier with a TCP/IP address.

soft link. CDS: A pointer that provides an alternative name for an object entry, directory, or other soft link in the name space. A soft link can be permanent or it can expire after a specific period of time. The CDS server also can delete it after the name that the link points to is deleted.

specific. XOM: The attribute types that can appear in an instance of a given class, but not in an instance of its superclasses.

SPI. (1) System programming interface. (2) Stub programming interface.

SRT. Structure rule table.

standard. A model that is established and widely used.

string. An ordered sequence of bits, octets, or characters, accompanied by the string's length.

structure rule table (SRT). GDS: A recurring attribute of the directory schema with the description of the permitted structures of distinguished names.

stub. RPC: A code module specific to an RPC interface that is generated by the Interface Definition Language (IDL) compiler to support remote procedure calls for the interface. RPC stubs are linked with client and server applications and hide the intricacies of remote procedure calls from the application code. See *client stub* and *server stub*.

Stub programming interface (SPI). RPC : A private runtime interface whose routines are unavailable to application code.

subclass. GDS, XOM: One of the classes whose attribute types are a superset of those of another class.

subject identifier (SID). A string that identifies a user or set of users. Each SID consists of three fields in the form person.group.organization. In an account, each field must have a specific value; in an access control list (ACL) entry, one or more fields may use a wildcard.

subobject. XOM: An object that is in a subordinate relationship to a given object.

subordinate. GDS, XDS: In the directory information tree (DIT), an entry whose distinguished name includes that of the other as a prefix.

superclass. GDS, XOM: One of the classes, designated as such, whose attribute types are a subset of those of another class.

superior. XDS: In the directory information tree (DIT), an entry whose distinguished name is included as a prefix of the distinguished name of the other. Each entry has exactly one immediate superior.

superobject. XOM: An object that is in a superior relationship to a given object.

synchronization. DTS: The process by which a Distributed Time Service entity requests clock values from other systems, computes a new time from the values, and adjusts its system clock to the new time.

syntax. (1) XOM: An object management (OM) syntax is any of the various categories into which the OM specification statically groups values on the basis of their form. These categories are additional to the OM type of the value. (2) A category into which an attribute value is placed on the basis of its form. See *attribute syntax*.

syntax template. XOM: A lexical construct containing an asterisk from which several attribute syntaxes can be derived by substituting text for the asterisk.

System programming interface (SPI). A private interface reserved for use by other services within a system and not available to application code. Contrast with *API*.

system time. The time value maintained and used by the operating system.

T

TCP. Transmission Control Protocol

TCP/IP. Transmission Control Protocol/Internet Protocol

TDF. Time differential factor.

terminal-owning region (TOR). CICS/ESA: A CICS address space whose primary purpose is to manage terminals.

thread. A single sequential flow of control within a process.

thread handle. RPC: A data item that enables threads to share a storage management environment.

Threads Service. A DCE component that provides portable facilities that support concurrent programming. The threads service includes operations to create and control multiple threads of execution in a single process and to synchronize access to global data within an application.

ticket. Security: An application-transparent mechanism that transmits the identity of an initiating principal to its target. See *privilege ticket*, *service ticket*, *simple ticket* and *ticket-granting ticket*.

ticket-granting ticket. Security: A ticket to the ticket-granting service. See *privilege ticket*, *service ticket*, and *simple ticket*.

time differential factor (TDF). DTS: The difference between universal time coordinated (UTC) and the time in a particular time zone.

TOR. Terminal-owning region.

transfer syntax. RPC: A set of encoding rules used for transmitting data over a network and for converting application data to and from different local data representations. See also *Network Data Representation*.

Transmission Control Protocol (TCP). A communications protocol used in Internet and any other network following the U.S. Department of Defense standards for inter-network protocol. TCP provides a reliable host-to-host protocol in packet-switched communication networks and in an interconnected system of such networks. It assumes that the Internet Protocol is the underlying protocol. The protocol that provides a reliable, full-duplex, connection-oriented service for applications.

Transmission Control Protocol/Internet Protocol (TCP/IP). A set of non-proprietary communications protocols that support peer-to-peer connectivity functions for both local and wide area networks.

transport layer. A network service that provides end-to-end communications between two parties, while hiding the details of the communications network. The Transmission Control Protocol (TCP) and International Organization for Standardization (ISO) TP4 transport protocols provide full-duplex virtual circuits on which delivery is reliable, error free, sequenced, and duplicate free. User Datagram Protocol (UDP) provides no guarantees. The connectionless RPC protocol provides some guarantees on top of UDP.

transport protocol. A communications protocol, such as the Transmission Control Protocol (TCP) or the User Datagram Protocol (UDP).

type. XOM: A category into which attribute values are placed on the basis of their purpose. See *attribute type*.

type UUID. RPC: The Universal Unique Identifier (UUID) that identifies a particular type of object and an associated manager. See also *manager* and *object*.

U

UDP. User Datagram Protocol.

unexport. RPC: To remove binding information from a server entry in a name service database. Contrast with *export*.

Universal Time Coordinated (UTC). The basis of standard time throughout the world. Synonymous with Greenwich mean time (GMT).

Universal Unique Identifier (UUID). RPC: An identifier that is immutable and unique across time and space. A UUID can uniquely identify an entity such as

an object or an RPC interface. See *interface UUID*, *object UUID*, and *type UUID*.

unmarshalling. RPC: The process by which a stub disassembles incoming network data and converts it into local data in the appropriate local data representation. Contrast with *marshalling*.

update timestamp (UTS). CDS: An attribute that identifies the time at which the most recent change was made to any attribute of a particular CDS name. For directories, the UTS reflects changes made only to attributes that apply to the actual directory (not one of its replicas).

user. A person who requires the services of a computing system.

User Datagram Protocol (UDP). In TCP/IP, a packet-level protocol built directly on the Internet protocol layer. UDP is used for application-to-application programs between TCP/IP host systems.

UTC. Universal Time Coordinated

UTS. Update timestamp.

UUID. Universal unique identifier

V

value. XOM: An arbitrary and complex information item that can be viewed as a characteristic or property of an object. See *attribute value*.

vector. RPC: An array of references to other structures.

vendor. Supplier of software products.

W

well-known endpoint. RPC: A preassigned, stable endpoint that a server can use every time it runs. Well-known endpoints typically are assigned by a central authority responsible for a transport protocol. An application declares a well-known endpoint either as an attribute in an RPC interface header or as a variable in the server application code. See *dynamic endpoint* and *endpoint*.

work space. XDS, XOM: A space in which OM objects of certain OM classes can be created, together with an implementation of the object management functions that supports those OM classes.

workstation. A device that enables users to transmit information to or receive information from a computer, for example, a display station or printer.

X

X.500. The CCITT/ISO standard for the open systems interconnection (OSI) application-layer directory. It allows users to register, store, search, and retrieve information about any objects or resources in a network or distributed system.

XDS. The X/Open Directory Services API.

X/Open Directory Services (XDS). An application program interface that DCE uses to access its directory service components. XDS provides facilities for adding, deleting, and looking up names and their attributes. The XDS library detects the format of the name to be looked up and directs the calls it receives to either GDS or CDS. XDS uses the XOM API to define and manage its information.

XOM. The X/Open OSI-Abstract-Data Manipulation API.

Bibliography

This bibliography is a list of publications for z/OS DCE and other products. The complete title, order number, and a brief description is given for each publication.

z/OS DCE Publications

This section lists and provides a brief description of each publication in the z/OS DCE library.

Overview

- *z/OS DCE Introduction*, GC24-5911

This book introduces z/OS DCE. Whether you are a system manager, technical planner, z/OS system programmer, or application programmer, it will help you understand DCE and evaluate the uses and benefits of including z/OS DCE as part of your information processing environment.

Planning

- *z/OS DCE Planning*, GC24-5913

This book helps you plan for the organization and installation of z/OS DCE. It discusses the benefits of distributed computing in general and describes how to develop plans for a distributed system in a z/OS environment.

Administration

- *z/OS DCE Configuring and Getting Started*, SC24-5910

This book helps system and network administrators configure z/OS DCE.

- *z/OS DCE Administration Guide*, SC24-5904

This book helps system and network administrators understand z/OS DCE and tells how to administer it from the batch, TSO, and shell environments.

- *z/OS DCE Command Reference*, SC24-5909

This book provides reference information for the commands that system and network administrators use to work with z/OS DCE.

- *z/OS DCE User's Guide*, SC24-5914

This book describes how to use z/OS DCE to work with your user account, use the directory service,

work with namespaces, and change access to objects that you own.

Application Development

- *z/OS DCE Application Development Guide: Introduction and Style*, SC24-5907

This book assists you in designing, writing, compiling, linking, and running distributed applications in z/OS DCE.

- *z/OS DCE Application Development Guide: Core Components*, SC24-5905

This book assists programmers in developing applications using application facilities, threads, remote procedure calls, distributed time service, and security service.

- *z/OS DCE Application Development Guide: Directory Services*, SC24-5906

This book describes the z/OS DCE directory service and assists programmers in developing applications for the cell directory service and the global directory service.

- *z/OS DCE Application Development Reference*, SC24-5908

This book explains the DCE Application Program Interfaces (APIs) that you can use to write distributed applications on z/OS DCE.

Reference

- *z/OS DCE Messages and Codes*, SC24-5912

This book provides detailed explanations and recovery actions for the messages, status codes, and exception codes issued by z/OS DCE.

z/OS SecureWay® Security Server Publications

This section lists and provides a brief description of books in the z/OS SecureWay Security Server library that may be needed for z/OS SecureWay Security Server DCE and for RACF® interoperability.

- *z/OS SecureWay Security Server DCE Overview*, GC24-5921

This book describes the z/OS SecureWay Security Server DCE and provides z/OS SecureWay Security Server DCE information about the z/OS DCE library.

- *z/OS SecureWay Security Server LDAP Client Programming*, SC24-5924

This book describes the Lightweight Directory Access Protocol (LDAP) client APIs that you can use to write distributed applications on z/OS DCE and gives you information on how to develop LDAP applications.

- *z/OS SecureWay Security Server RACF Security Administrator's Guide*, SA22-7683.

This book explains RACF concepts and describes how to plan for and implement RACF.

- *z/OS SecureWay Security Server LDAP Server Administration and Use*, SC24-5923

This book describes how to install, configure, and run the LDAP server. It is intended for administrators who will maintain the server and database.

- *z/OS SecureWay Security Server Firewall Technologies*, SC24-5922

This book provides the configuration, commands, messages, examples and problem determination for the z/OS Firewall Technologies. It is intended for network or system security administrators who install, administer and use the z/OS Firewall Technologies.

Tool Control Language Publication

- *Tcl and the Tk Toolkit*, John K. Osterhout, (c)1994, Addison—Wesley Publishing Company.

This non-IBM book on the Tool Control Language is useful for application developers, DCECP script writers, and end users.

IBM C/C++ Language Publication

- *z/OS C/C++ Programming Guide*, SC09-4765

This book describes how to develop applications in the C/C++ language in z/OS.

z/OS DCE Application Support Publications

This section lists and provides a brief description of each publication in the z/OS DCE Application Support library.

- *z/OS DCE Application Support Configuration and Administration Guide*, SC24-5903

This book helps system and network administrators understand and administer Application Support.

- *z/OS DCE Application Support Programming Guide*, SC24-5902

This book provides information on using Application Support to develop applications that can access CICS® and IMS™ transactions.

Encina Publications

- *z/OS Encina Toolkit Executive Guide and Reference*, SC24-5919

This book discusses writing Encina applications for z/OS.

- *z/OS Encina Transactional RPC Support for IMS*, SC24-5920

This book is to help software designers and programmers extend their IMS transaction applications to participate in a distributed, transactional client/server application.

Index

A

abstract OM class 112, 113
Abstract Service 233
Abstract Service Definition 143
Abstract Syntax Notation One (ASN.1)
 abstract syntaxes 93
 ASN.1 simple types 94
 ASN.1 types 94
 encoding 348
 relating to Basic Encoding Rules 93
 sample definition 91
 transfer syntaxes 93
abstract syntaxes 93
add attribute 256
address 245
ADMD (administration management domain)
 See administration management domain (ADMD)
administration management domain (ADMD) 302
administrative limit exceeded 266
alias entry 85
API (Application Program Interface)
 See Application Program Interface (API)
Application Program Interface (API)
 XDS 234
approximate match 260
arbitrary encoding 348
arguments 233
ASN.1 (Abstract Syntax Notation One)
 See Abstract Syntax Notation One (ASN.1)
AT (attribute table)
 See attribute table (AT)
attribute 245, 247
 cacheable 318
 error 246
 list 246
 matching rules 142
 multi-valued 142
 OM syntax 122
 enumerated type 123
 object type 123
 string type 124
 OM value syntax 142
 selected 255
 syntax template 122
 type 246, 333, 339, 340
 value 122
 value length 142
 values 246
attribute table (AT) 92
attribute type
 directory 79

attribute type (*continued*)
 mandatory 91
 OM 97
 optional 91
attribute value assertion (AVA) 248
 multiple AVAs 84
 relationship to RDN 84
audience skills xvii
automatic connection management 141
automatic continuation 252
AVA (attribute value assertion) 248, 254

B

Basic Directory Contents package 118
Basic Encoding Rules (BER) 95
bibliography 373
bit string 326
book organization xix
books, list of DCE and related 373
Boolean 340

C

C programming language
 interface to 325
 naming conventions
 XDS and XOM APIs 100
canonical-ber 347
CCITT 348
chaining prohibited 251
character set 333
character string 326, 328, 338, 341
class 347, 349
 abstract OM class 112
 concrete OM class 112
 OM class hierarchy 112
 OM class inheritance 112
 OM object 111
closure, package 120
common results 248
communications error 248
compare result 249
concrete OM class 112, 113
context 142, 233, 249
 common arguments 142
 local controls 142
 service controls 142
continuation reference 240, 252

D

data

type

Boolean 333

data value descriptor 348

DC_C_SESSION 140

DDA (domain-defined attribute)

See domain-defined attribute (DDA)

default

context 236, 252

session 235, 272

descriptor list 102

initializing 126

OM_descriptor data structure 102, 125

representation of public object 103

descriptor, service-generated 334

descriptors 333

direct reference 349

directory

alias entry 85

attribute table 92

attribute type 79

attribute types

mandatory 91

optional 91

automatic connection management 141

building a distinguished name 105

context 142

defining subclasses 93

DIB (Directory Information Base) 78

Directory Class Definitions 142

Directory Service Package 142

distinguished name 83, 105

ds_add_entry 151

ds_list 151

ds_remove_entry 151

example of directory entry 81

filter 150

GDS standard schema 87

modify operations 150

modifying entries 151

name verification 86

naming attributes 88

Object Class Table 89

object entries 81

object identifiers 80

objects 78

reading an entry 144

relationship between schemas and the DIT 93

relative distinguished name (RDN) 84

search criteria 150

selected attribute types 142

selected object classes 142

session 140

structure of the DIB 78

directory (*continued*)

Structure Rule Table (SRT) 87

Directory Class Definitions 142

Directory class, DL-Submit-Permission 310

Directory Connection Management Functions 137, 140

directory information base, schema 83

Directory Information Model 78

directory information tree

example of a distinguished name 83

GDS standard schema 87

relationship between schemas and the DIT 93

directory modify operations 150

ds_add_entry 150

ds_modify_entry 150

ds_modify_rdn 150

ds_remove_entry 150

Directory operation functions 143

directory read operations 143

directory search operations 150

Directory Service

functions 137

Package 142

Directory System Agent (DSA)

address 245, 272

name

XDS attribute 272

distinguished encoding 237

distinguished name 83

as a public object 105

building 105

example of distinguished name 83

structure 105

distribution list (DL) 297

DL (distribution list)

See distribution list (DL)

DL-Submit-Permission 310

DMD 251

domain-defined attribute (DDA) 307

DS_A_ALIASED_OBJECT_NAME 279

DS_A_BUSINESS_CATEGORY 279

DS_A_COMMON_NAME 279

DS_A_COUNTRY_NAME 279

DS_A_DELIV_CONTENT_LENGTH 298

DS_A_DELIV_CONTENT_TYPES 298

DS_A_DELIVERABLE_EITS 298

DS_A_DESCRIPTION 279

DS_A_DESTINATION_INDICATOR 279

DS_A_DL_MEMBERS 299

DS_A_DL_SUBMIT_PERMS 299

DS_A_FACSIMILE_TELEPHONE_NUMBER 279

DS_A_INTERNATIONAL_ISDN_NUMBER 279

DS_A_KNOWLEDGE_INFORMATION 280

DS_A_LOCALITY_NAME 280

DS_A_MEMBER 280

DS_A_MESSAGE_STORE 299
DS_A_OBJECT_CLASS 280
DS_A_OR_ADDRESSES 299
DS_A_ORGANIZATION_NAME 280
DS_A_ORGANIZATIONAL_UNIT_NAME 280
DS_A_OWNER 280
DS_A_PHYSICAL_DELIVERY_OFFICE_NAME 280
DS_A_POST_OFFICE_BOX 280
DS_A_POSTAL_ADDRESS 280
DS_A_POSTAL_CODE 280
DS_A_PREF_DELIV_METHODS 299
DS_A_PREFERRED_DELIVERY_METHOD 280
DS_A_PRESENTATION_ADDRESS 281
DS_A_REGISTERED_ADDRESS 281
DS_A_ROLE_OCCUPANT 281
DS_A_SEARCH_GUIDE 281
DS_A_SEE_ALSO 281
DS_A_SERIAL_NUMBER 281
DS_A_STATE_OR_PROVINCE_NAME 281
DS_A_STREET_ADDRESS 281
DS_A_SUPP_AUTO_ACTIONS 299
DS_A_SUPP_CONTENT_TYPES 299
DS_A_SUPP_OPT_ATTRIBUTES 299
DS_A_SUPPORTED_APPLICATION_CONTEXT 281
DS_A_SURNAME 282
DS_A_TELEPHONE_NUMBER 282
DS_A_TELETEX_TERMINAL_IDENTIFIER 282
DS_A_TELEX_NUMBER 282
DS_A_TITLE 282
DS_A_USER_PASSWORD 282
DS_A_X121_ADDRESS 282
ds_abandon 137
DS_ADD_ATTRIBUTE 256
ds_add_entry 150, 155
DS_ADD_VALUES 256
DS_ADMINISTRATIVE_LIMIT_EXCEEDED 266
DS_AE_TITLE 245
DS_ALIAS_DEREFERENCED 248
DS_ALIAS_ENTRY 263
DS_ALIASED_RDNS 253
DS_ALL_ATTRIBUTES 255
DS_AND 259
DS_ANY_DELIVERY_METHOD 281
DS_APPROXIMATE_MATCH 260
DS_ASYNCHRONOUS 252
DS_ATTRIBUTE_TYPE 246, 247, 276, 285
DS_ATTRIBUTE_VALUE 247
DS_ATTRIBUTE_VALUES 246, 276
DS_ATTRIBUTES_SELECTED 255
DS_AUTOMATIC_CONTINUATION 252
DS_AVAS 254
DS_BASIC_DIRECTORY_CONTENTS_PACKAGE 275
ds_bind 127, 140
 automatic connection management 141
DS_C_ABANDON_FAILED 244
DS_C_ACCESS_POINT 245
DS_C_ADDRESS 245
DS_C_ATTRIBUTE 245, 276
DS_C_ATTRIBUTE_ERROR 246
DS_C_ATTRIBUTE_LIST 246
DS_C_ATTRIBUTE_PROBLEM 265
DS_C_ATTRIBUTES 247
DS_C_AVA 248
DS_C_COMMON_RESULTS 248
DS_C_COMMUNICATIONS_ERROR 248
DS_C_COMPARE_RESULT 249
DS_C_CONTEXT 142, 249
DS_C_CONTINUATION_REF 252
DS_C_DS_DN 253
DS_C_DS_RDN 253
DS_C_ENTRY_INFO 254
DS_C_ENTRY_INFO_SELECTION 254
DS_C_ENTRY_MODIFICATION 255
DS_C_ENTRY_MODIFICATION_LIST 256
DS_C_ERROR 239, 256
DS_C_EXTENSION 258
DS_C_FACSIMILE_TELEPHONE_NUMBER 284
DS_C_FILTER 259
DS_C_FILTER_ITEM 260
DS_C_LIBRARY_ERROR 261
DS_C_LIST_INFO 262
DS_C_LIST_INFO_ITEM 263
DS_C_LIST_RESULT 156, 264
DS_C_NAME 264
DS_C_NAME_ERROR 265
DS_C_OPERATION_PROGRESS 265
DS_C_PARTIAL_OUTCOME_QUAL 266
DS_C_POSTAL_ADDRESS 284
DS_C_PRESENTATION_ADDRESS 267
DS_C_READ_RESULT 148, 268
DS_C_REFERRAL 268
DS_C_RELATIVE_NAME 268
DS_C_SEARCH_CRITERION 285
DS_C_SEARCH_GUIDE 285, 286
DS_C_SEARCH_INFO 268, 269
DS_C_SEARCH_RESULT 269
DS_C_SECURITY_ERROR 270
DS_C_SESSION 271
DS_C_SYSTEM_ERROR 272
DS_C_TELETEX_TERMINAL_IDENTIFIER 286
DS_C_TELEX_NUMBER 287
DS_C_UPDATE_ERROR 273
DS_CHAINING_PROHIBITED 251
DS_CHANGES 256
ds_compare 143, 147
ds_compare, DS_C_COMPARE_RESULT 147
DS_COMPLETED 266
DS_COUNTRY 251
DS_COUNTRY_CODE 287
DS_CRIT 258

DS_CRITERIA 285, 286
 DS_DEFAULT_CONTEXT 236, 252, 320
 DS_DEFAULT_SESSION 140, 235, 272, 321
 DS_DMD 251
 DS_DN 253
 DS_DONT_DEREFERENCE_ALIASES 251
 DS_DONT_USE_COPY 251
 DS_DSA_ADDRESS 272
 DS_DSA_NAME 272
 DS_E_ADMINISTRATIVE_LIMIT_EXCEEDED 270
 DS_E_AFFECTS_MULTIPLE_DSAS 273
 DS_E_ALIAS_DEREFERENCING_PROBLEM 265
 DS_E_ALIAS_PROBLEM 265
 DS_E_BAD_ARGUMENT 261
 DS_E_BAD_CLASS 262
 DS_E_BAD_CONTEXT 262
 DS_E_BAD_NAME 262
 DS_E_BAD_SESSION 262
 DS_E_BUSY 270
 DS_E_CANNOT_ABANDON 244
 DS_E_COMMUNICATIONS_PROBLEM 249
 DS_E_CONSTRAINT_VIOLATION 247
 DS_E_DIT_ERROR 271
 DS_E_ENTRY_ALREADY_EXISTS 273
 DS_E_INAPPROP_AUTHENTICATION 270
 DS_E_INAPPROP_MATCHING 247
 DS_E_INSUFFICIENT_ACCESS 270
 DS_E_INVALID_ATTRIBUTE_SYNTAX 247
 DS_E_INVALID_ATTRIBUTE_VALUE 265
 DS_E_INVALID_CREDENTIALS 270
 DS_E_INVALID_REFERENCE 271
 DS_E_INVALID_SIGNATURE 270
 DS_E_LOOP_DETECTED 271
 DS_E_MISCELLANEOUS 262
 DS_E_MISSING_TYPE 262
 DS_E_MIXED_SYNCHRONOUS 262
 DS_E_NAMING_VIOLATION 273
 DS_E_NO_INFORMATION 270
 DS_E_NO_SUCH_ATTRIBUTE_OR_VALUE 247
 DS_E_NO_SUCH_OBJECT 265
 DS_E_NO_SUCH_OPERATION 244
 DS_E_NOT_ALLOWED_ON_NON_LEAF 273
 DS_E_NOT_ALLOWED_ON_RDN 273
 DS_E_NOT_SUPPORTED 262
 DS_E_OBJECT_CLASS_VIOLATION 273
 DS_E_OUT_OF_SCOPE 271
 DS_E_PROTECTION_REQUIRED 270
 DS_E_TIME_LIMIT_EXCEEDED 271
 DS_E_TOO_LATE 244
 DS_E_TOO_MANY_OPERATIONS 262
 DS_E_TOO_MANY_SESSIONS 262
 DS_E_UNABLE_TO_PROCEED 271
 DS_E_UNAVAILABLE 271
 DS_E_UNAVAILABLE_CRITICAL_EXTENSION 271
 DS_E_UNDEFINED_ATTRIBUTE_TYPE 247
 DS_E_UNWILLING_TO_PERFORM 271
 DS_ENTRIES 269
 DS_ENTRY 268
 DS_EQUALITY 261
 DS_EXT 250
 DS_FILE_DESCRIPTOR 272
 DS_FILTER_ITEM_TYPE 260, 285
 DS_FILTER_ITEMS 259
 DS_FILTER_TYPE 259, 285
 DS_FILTERS 259
 DS_FINAL_SUBSTRING 261
 DS_FROM_ENTRY 249, 254, 263
 DS_G3_FACSIMILE_DELIVERY 281
 DS_G4_FACSIMILE_DELIVERY 281
 DS_GREATER_OR_EQUAL 261
 DS_HIGH 251
 DS_IA5_TERMINAL_DELIVERY 281
 DS_IDENTIFIER 259
 DS_INDIVIDUAL 310
 DS_INFORMATION_TYPE 255
 DS_INITIAL_SUBSTRING 261
 ds_initialize 127, 138
 DS_ITEM_PARAMETERS 259
 DS_LESS_OR_EQUAL 261
 DS_LIMIT_PROBLEM 266
 ds_list 147, 150, 155
 DS_LIST_INFO 264
 ds_list, DS_C_LIST_RESULT 147, 156
 DS_LOCAL_SCOPE 251
 DS_LOW 251
 DS_MATCHED 249, 265
 DS_MAX_OUTSTANDING_OPERATIONS 239
 DS_MEDIUM 251
 DS_MEMBER_OF_DL 311
 DS_MEMBER_OF_GROUP 311
 DS_MHS_DELIVERY 281
 DS_MODIFICATION_TYPE 255
 ds_modify_entry 150
 ds_modify_rdn 150
 DS_N_ADDRESSES 267
 DS_NAME_RESOLUTION_PHASE 266
 DS_NEXT_RDN_TO_BE_RESOLVED 266
 DS_NO_LIMIT_EXCEEDED 266
 DS_NOT 260
 DS_NOT_STARTED 266
 DS_O_ALIAS 283
 DS_O_APPLICATION_ENTITY 283
 DS_O_APPLICATION_PROCESS 283
 DS_O_COUNTRY 283
 DS_O_DEVICE 283
 DS_O_DSA 283
 DS_O_GROUP_OF_NAMES 283
 DS_O_LOCALITY 283
 DS_O_ORGANIZATION 283
 DS_O_ORGANIZATIONAL_PERSON 283

DS_O_ORGANIZATIONAL_ROLE 283
DS_O_ORGANIZATIONAL_UNIT 283
DS_O_PERSON 283
DS_O_RESIDENTIAL_PERSON 283
DS_O_TOP 280, 282
DS_OBJECT_CLASS 286
DS_OBJECT_NAME 246, 249, 254, 263, 269
DS_OPERATION_NOT_STARTED 250, 266
DS_OPERATION_PROGRESS 250, 253
DS_OR 260
DS_P_SELECTOR 267
DS_PARAMETERS 284, 286
DS_PARTIAL_OUTCOME_QUALIFIER 263, 269
DS_PATTERN_MATCH 311
DS_PERFORMER 248
DS_PERM_TYPE 310
DS_PHYSICAL_DELIVERY 281
DS_POSTAL_ADDRESS 284
DS_PREFER_CHAINING 251
DS_PRESENT 261
DS_PRIORITY 251
DS_PROBLEM 244, 247, 256, 265, 270, 273
DS_PROCEEDING 266
DS_RDN 253, 263
DS_RDNS_RESOLVED 253
ds_read 127
 completion status 147
 data structure 111
 distinguished name 132
 DS_C_READ_RESULT 147, 148
 functions implemented 143
 input 111
 input parameters 102
 name parameter 109
 OM class returned 147
 output 148
 parameter to 112
 performing a read operation 147
 pointer returned 121
 private objects 145
 public object 145
 code example 99
 read parameter 131
 read result 148
 reading an entry 144
 returning a pointer 120
 selection parameter 146
 summary of OM calls 128
ds_receive_result 137
DS_REMOVE_ATTRIBUTE 256
ds_remove_entry 150
DS_REQUESTOR 272
DS_S_SELECTOR 267
DS_SCOPE_OF_REFERRAL 251
ds_search 147, 150
 DS_C_SEARCH_RESULT 147
 ds_search (*continued*)
 filter 150
DS_SELECT_ALL_TYPES 237
DS_SELECT_ALL_TYPES_AND_VALUES 238
DS_SELECT_NO_ATTRIBUTES 237
DS_SERVICE_PKG 241
ds_shutdown 140
DS_SIZE_LIMIT 251
DS_SIZE_LIMIT_EXCEEDED 267
DS_SUBORDINATES 263
DS_SUBSTRINGS 261
DS_SUCCESS 239, 241
DS_T_SELECTOR 267
DS_TARGET_OBJECT 253
DS_TELEPHONE_DELIVERY 281
DS_TELEPHONE_NUMBER 284
DS_TELETEX_DELIVERY 281
DS_TELETEX_TERMINAL 286
DS_TELEX_DELIVERY 281
DS_TIME_LIMIT 251
DS_TIME_LIMIT_EXCEEDED 252, 267
DS_TYPES_AND_VALUES 255
DS_TYPES_ONLY 255
DS_UNAVAILABLE_CRITICAL_EXTENSIONS 267
ds_unbind 141
ds_unbind, automatic connection management 141
DS_UNCORRELATED_LIST_INFO 264
DS_UNCORRELATED_SEARCH_INFO 269
DS_UNEXPLORED 267
ds_version 138
 Basic Directory Contents Package 138
 Directory Service Package 138
 Global Directory Service Extension Package 138
 MHS Directory User Package 138
 negotiating features 118
 package 118
 xds.h header file 138
 xdsbdcp.h header file 138
 xdsgds.h header file 138
 xdsmdup.h header file 138
 xmhp.h header file 138
 xmsga.h header file 138
DS_VIDEOTEX_DELIVERY 281
DSA (Directory System Agent)
 See Directory System Agent (DSA)
DSX_C_GDS_ACL 317
DSX_C_GDS_CONTEXT 318
DSX_C_GDS_SESSION 321
DSX_DIR_ID 321
DSX_DONT_STORE 318
DSX_DUA_CACHE 319
DSX_DUAFIRST 318
DSX_NORMAL_CLASS 318
DSX_PASSWORD 321
DSX_PRIV_CLASS 319

DSX_RESIDENT_CLASS 319
DSX_USED_SA 319

E

EIT (encoded information types)
See encoded information types (EIT)

elements 338

encoded
information
 encoded information type (EIT) 298
 type 298

encoded information types (EIT) 298

encoding 339

entries 268, 269

entry
information 254
 selection 254
modification 255
modification, list 256

Enum(*) 325

enumerated type 123

enumeration 334, 340

equality 261

example.c 159

exclusions 334

extensions 250, 258

external 348

F

facsimile telephone number 284

federated DCE namespace 9, 12

filter 150, 259
item 259
item type 260
type 259

final substring 261

from entry 249, 254, 263

G

GDS (Global Directory Service)
See Global Directory Service (GDS)

Global Directory Service (GDS)
context 318
DUA 321
session 321
standard schema 87
 attribute table 92
 naming attributes 88
 Object Class Table 89
 Structure Rule Table (SRT) 87
 structured object classes 88
XDS Application Programming Interface 77
XOM Application Programming Interface 77

Global Directory Service Extension Package 119

glossary 355

greater or equal 261

H

high priority 251

I

identifier 259

indirect reference 349

information type 255

initial substring 261

integer
OM 335, 340

interface
to C 325

intermediate data type 332

ISO (International Organization for Standardization) 348

item
filter 260
parameters 259

L

length-unspecified 338

less than or equal to, XDS 261

library error 261

limit problem 266

list
info 262, 264
info item 263
result 264

local scope 251

local-string 333

low priority 251

M

management domain (MD) 307

matched 249

max outstanding operations 239

MD (management domain)
See management domain (MD)

MDUP (MHS Directory User Package)
See MHS Directory User Package (MDUP)

medium priority 251

message handling system 297

message handling system (MHS) 297

message store (MS) 299

message transfer agent (MTA) 297

metacharacters 27
in CDS 27
in DNS 27
in GDS 27

MH class, OR-ADDRESS 300
MH_T_ADMD_NAME 302
MH_T_COMMON_NAME 302, 303
MH_T_DOMAIN_TYPE_1 303
MH_T_DOMAIN_TYPE_2 303
MH_T_DOMAIN_TYPE_3 303
MH_T_DOMAIN_TYPE_4 303
MH_T_DOMAIN_VALUE_1 303
MH_T_DOMAIN_VALUE_2 303
MH_T_DOMAIN_VALUE_3 303
MH_T_DOMAIN_VALUE_4 303
MH_T_GENERATION 303
MH_T_GIVEN_NAME 303
MH_T_INITIALS 303
MH_T_ISDN_NUMBER 303
MH_T_ISDN_SUBADDRESS 303
MH_T_NUMERIC_USER_IDENTIFIER 304
MH_T_ORGANIZATION_NAME 304
MH_T_ORGANIZATIONAL_UNIT_NAME_1 304
MH_T_ORGANIZATIONAL_UNIT_NAME_2 304
MH_T_ORGANIZATIONAL_UNIT_NAME_3 304
MH_T_ORGANIZATIONAL_UNIT_NAME_4 304
MH_T_POSTAL_ADDRESS_DETAILS 304
MH_T_POSTAL_ADDRESS_IN_FULL 304
MH_T_POSTAL_ADDRESS_IN_LINES 304
MH_T_POSTAL_CODE 304
MH_T_POSTAL_COUNTRY_NAME 304
MH_T_POSTAL_DELIV_SYSTEM_NAME 305
MH_T_POSTAL_DELIVERY_POINT_NAME 305
MH_T_POSTAL_GENERAL_DELIV_ADDRESS 305
MH_T_POSTAL_LOCALE 305
MH_T_POSTAL_OFFICE_BOX_NUMBER 305
MH_T_POSTAL_OFFICE_NAME 305
MH_T_POSTAL_OFFICE_NUMBER 305
MH_T_POSTAL_ORGANIZATION_NAME 305
MH_T_POSTAL_PATRON_DETAILS 305
MH_T_POSTAL_PATRON_NAME 305
MH_T_POSTAL_STREET_ADDRESS 305
MH_T_PRESENTATION_ADDRESS 305
MH_T_PRMD_NAME 306
MH_T_SURNAME 306
MH_T_TERMINAL_IDENTIFIER 306
MH_T_TERMINAL_TYPE 306
MHS (message handling system)
See message handling system (MHS)
MHS Directory User Package 119
MHS Directory User Package (MDUP) 297
Mnemonic O/R Address 308
modification parameter 335
modification type 255
MS (message store)
See message store (MS)
MTA (message transfer agent)
See message transfer agent (MTA)

N

name 264
error 265
maximum name sizes 29
naming rules 25
resolution phase 266
valid characters 25
namespace, federated DCE 9, 12
naming attributes 88
network addresses 267
no limit exceeded 266
Numeric O/R Address 308

O

O/R (originator/recipient)
See originator/recipient (O/R)
object 340
accessing in federated DCE namespace 12
class 347
class hierarchy 112
directory 78
dynamically defined static public 172
encoding 347
identifier 335
name
directory entry 246
distinguished, XDS 254
target 263
XDS attribute 249, 269
object type 123
OM 97
syntax 97
OM attribute type 97
OM class 111
partially defined static public 171
predefined static public 170
private
attribute type 340
declaration 335
XOM 337
public
attribute type 340
declaration 335
description 102
descriptor 333
representation of public object 103
selected attribute types 142
selected object classes 142
subordinate 263
subordinate object 126
value 97
object class attribute 81
Object Class Table (OCT)
acronyms of super class 89

Object Class Table (OCT) *(continued)*

- class inheritance 89
- description 89
- mandatory attributes 91
- optional attributes 91
- partial representation of the OCT 89

object entry

- class attribute 81
- directory 81
- example of directory entry 81
- reading an entry 144

object identifier

- directory 80
- Object Class Table 90
- OM class 100
- XDS Directory Service Package 80

object OM class inheritance 112**Object(*) syntax template** 325**OCT (Object Class Table)**

- See Object Class Table (OCT)

octet

- aligned encoding 349
- string 326

OM attribute types

- mandatory 100
- optional 100

OM class

- abstract 112
- concrete 112
- defining 115
- hierarchy 112
- inheritance 112
- initial value 116
- mapping the class definition 98
- object identifier 100
- OM attribute 116
- value length 116
- value number 116
- value syntax 116

OM objects 97**OM value syntax** 142**OM_C_SERVICE_ERROR** 270**om_copy_value()** 128, 342**om_copy()** 128, 342**om_create()** 128, 342**om_decode()** 128, 342**om_delete** 110**om_delete()** 128, 342**OM_descriptor** 125**om_encode()** 128, 342**OM_ENCODING_INVALID** 344**OM_EXPORT** 133, 134, 144**OM_FUNCTION_DECLINED** 344**OM_FUNCTION_INTERRUPTED** 344**om_get** 109, 110, 111, 120, 121, 128, 130, 131

- exclusions parameter 130

om_get *(continued)*

- extracting data from DS_C_READ_RESULT 131
- input parameters 130

om_get() 128, 342**OM_IMPORT** 133, 134**om_instance()** 128, 342**OM_MEMORY_INSUFFICIENT** 344**OM_NETWORK_ERROR** 344**OM_NO_SUCH_CLASS** 344**OM_NO_SUCH_EXCLUSION** 344**OM_NO_SUCH_MODIFICATION** 344**OM_NO_SUCH_OBJECT** 344**OM_NO_SUCH_RULES** 344**OM_NO_SUCH_SYNTAX** 344**OM_NO_SUCH_TYPE** 344**OM_NO_SUCH_WORKSPACE** 344**OM_NOT_AN_ENCODING** 344**OM_NOT_CONCRETE** 344**OM_NOT_PRESENT** 344**OM_NOT_PRIVATE** 344**OM_NOT_THE_SERVICES** 344**OM_NULL_DESCRIPTOR** 135**OM_OID_DESC** 134**OM_PERMANENT_ERROR** 344**OM_POINTER_INVALID** 344**om_put()** 128, 342**om_read()** 128, 342**om_remove()** 128, 342**OM_S_BOOLEAN** 325**OM_S_ENCODING** 339**OM_S_ENUMERATION** 329, 339**OM_S_GENERAL_STRING** 339**OM_S_GENERALISED_TIME_STRING** 339**OM_S_GRAPHIC_STRING** 339**OM_S_IA5_STRING** 339**OM_S_INTEGER** 325, 339**OM_S_NULL** 325, 339**OM_S_NUMERIC_STRING** 339**OM_S_OBJECT_DESCRIPTOR_STRING** 339**OM_S_OBJECT_IDENTIFIER_STRING** 339**OM_S_OCTET_STRING** 339**OM_S_PRINTABLE_STRING** 339**OM_S_TELETEX_STRING** 339**OM_S_UTC_TIME_STRING** 339**OM_S_VIDEOTEX_STRING** 339**OM_S_VISIBLE_STRING** 339**OM_sint** 332**OM_sint16** 332**OM_sint32** 332**OM_STRING** 135, 338**OM_SUCCESS** 344**OM_SYSTEM_ERROR** 344**OM_TEMPORARY_ERROR** 344**OM_TOO_MANY_VALUES** 344**OM_uint** 332

OM_uint16 332
OM_uint32 332
OM_VALUES_NOT_ADJACENT 345
om_write() 128, 342
OM_WRONG_VALUE_LENGTH 345
OM_WRONG_VALUE_MAKEUP 345
OM_WRONG_VALUE_NUMBER 345
OM_WRONG_VALUE_POSITION 345
OM_WRONG_VALUE_SYNTAX 345
OM_WRONG_VALUE_TYPE 345
Open Systems Interconnection (OSI)
 application contexts 282
 application entity 245, 281
 communications 245
 presentation address 281
operation
 Directory service 233
 not started 250, 266
 performer 248
 progress 250, 253, 265
OR-NAME 310
originator/recipient (O/R) 299
OSI (Open Systems Interconnection)
 See Open Systems Interconnection (OSI)

P

package 117
 Basic Directory Contents 117, 118, 138
 closure 120
 Directory Service 117, 138
 ds_version 118
 Global Directory Service 117
 Global Directory Service Extension 119, 138
 MHS Directory User 117, 119, 138
 negotiating features 118
 Strong Authentication 119
 Strong Authentication Package 117
partial outcome qualifier 263, 266, 269
postal address 284
Postal O/R Address 308
preface xvii
prefer chaining 251
presentation
 address 267
 selector 267
priority 251
private management domain 302
private management domain, PRMD 302
private object
 comparison with public objects 111
 XDS, description 111
private representation 337
PRMD 302
public object 102
 client-generated 109

public object (*continued*)
 comparison with private objects 111
 creating 145
 dynamically defined static 172
 partially defined static 171
 predefined static 170
 representation using descriptor list 103
 service-generated 109

R

RDN (Relative Distinguished Name)
 See ?
read result 148, 268
referral 240, 268
relative distinguished name (RDN)
 determining sequence 253
 distinguishing an entry 84
 relationship to AVA 84
 resolving 253
 uniqueness of 84
relative name 268
remove attribute 256
requestor 272
return codes 337, 343
rules
 object encoding attribute 348

S

scope of referral
 XDS attribute 251
search
 criterion 285
 guide 286
 info 268
 result 269
security
 error, OM class 270
selected attribute type 275
sequence 329
service
 controls 251, 318
 error 270
 package 241
service controls 319
service interface data types 124
session 140, 233, 271
 DC_C_SESSION 140
 DS_DEFAULT_SESSION 140
 multiple concurrent 140
 selector 267
set 329
size limit
 exceeded 267
 maximum objects 251

skills, for audience xvii

standards
 Directory 233
 Directory OM classes 241

status
 directory service 233
 DS_status, return value 239

storage
 management, XOM API 121

string
 length, XOM 338
 octet and bit 326
 syntax 340

string type
 OM attribute 124

string(*) syntax template 326

Strong Authentication Package 119

structure of book xix

Structure Rule Table (SRT)
 GDS standard schema 87
 naming attributes 88
 structured object classes 88

structured object classes 88

structured postal address 308

subclass 282

substring 261

superclass
 DS_O_TOP 282
 OM 283

superclasses, OM 291

syntax template 122

system error 272

T

target object 253

teldir.c 170

teldir2.c 217

teletex terminal identifier 286

telex number 287

Terminal O/R Address 309

terminology
 in this book xix

time limit
 exceeded 252, 267
 XDS attribute 251

transfer syntax 93
 ASN.1 93

transport selector 267

type
 only 255
 values 255

U

unavailable critical extensions 267

uncorrelated
 list info 264
 search info 269

unexplored
 directory attribute 267

unstructured postal address 308

update error 273

V

value 334, 340
 length 341
 position 341

W

workspace 97, 120, 341

X

X.500 Directory Information Model 78

X.500, naming concepts 83

X121_ADDRESS 306

XDS API 77

XDS Application Interface 137
 acl.c 159
 acl.h 159
 automatic connection management 141
 Basic Directory Contents Package 117, 138
 C naming conventions 100
 client-generated public objects 109
 context 142
 data types for function calls 127
 DC_C_SESSION 140
 Directory Class Definitions 142
 Directory Connection Management Functions 137, 140
 directory modify operations 150
 Directory operation functions 143
 directory read operations 143
 directory search operations 150
 Directory Service 117
 Directory Service functions 137
 Directory Service Package 138
 ds_abandon 137
 ds_add_entry 151, 155
 ds_bind 127, 140
 DS_C_CONTEXT 142
 DS_C_LIST_RESULT 156
 ds_compare 143, 147
 DS_DEFAULT_SESSION 140
 ds_initialize 127, 138
 ds_list 147, 150, 151, 155
 ds_read
 code example 147

XDS Application Interface *(continued)*

- ds_read *(continued)*
 - completion status 147
 - data structure 111
 - distinguished name 132
 - functions implemented 143
 - input 111
 - input parameters 102
 - name parameter 109
 - OM class returned 147
 - output 148
 - parameter to 112
 - pointer returned 121
 - private objects 145
 - public object 99, 145
 - read parameter 131
 - returning a pointer 120
 - selection parameter 146
 - summary of OM calls 128
 - ds_receive_result 137
 - ds_remove_entry 151
 - ds_search 147, 150
 - ds_shutdown 140
 - ds_unbind 141
 - ds_version 138
 - dynamically defined static public objects 172
 - example.c 159
 - example.h 159
 - filter 150
 - Global Directory Service 117
 - Global Directory Service Extension Package 138
 - MHS Directory User Package 117, 138
 - modifying entries 151
 - multiple concurrent sessions 140
 - partially defined static public objects 171
 - performing a read operation 147
 - predefined static public objects 170
 - private objects 111
 - programming guidelines 159
 - sample programs 159, 170
 - session 140
 - Strong Authentication Package 117
 - teldir.c 159, 170
 - XDS Interface Class Definitions 141
 - XDS Interface Management Functions 137
 - xds.h header file 138
 - xdsbdcp.h header file 138
 - xdsgds.h header file 138
 - xdsmdup.h header file 138
 - xmhp.h header file 138
 - xmsga.h header file 138
- XDS Application Programming Interface 77**
XDS Directory Service Package, object identifier 80
XDS Interface Class Definitions 141
XDS Interface Management Functions 137

xds.h 241

xdsbdcp.h 275

XOM API 77

XOM Application Interface

- abstract OM class 112
- ASN.1 100
- C naming conventions 100
- client-generated public objects 109
- data types for function calls 127
- defining an OM class 115
- description 97
- enumerated type 123
- header files 133
- initializing descriptors 126
- macros 133, 144
- object identifier 100
- object type 123
- OM
 - attributes 97
 - class 111
 - class inheritance 112
 - concrete class 112
 - functions 128
 - objects 97
- OM attribute type 97
- OM syntax 97
- om_copy 120
- om_copy_value() 128
- om_copy() 128
- om_create() 128
- om_decode() 128
- om_delete 110
- om_delete() 128
- OM_descriptor data structure 125
- OM_descriptor structure 102
- om_encode() 128
- OM_EXPORT macro 133, 134, 144
- om_get 109, 110, 111, 120, 121, 128, 130, 131
- om_get () 130
- om_get() 128
- OM_IMPORT macro 133, 134
- om_instance() 128
- OM_NULL_DESCRIPTOR macro 135
- OM_OID_DESC macro 134
- om_put() 128
- om_read() 128
- om_remove() 128
- OM_STRING macro 135
- om_write() 128
- package closure 120
- private objects 111
- public object
 - creating 145
 - description 102
 - representation of 103
- return codes 133

XOM Application Interface *(continued)*

service interface data types 124

storage management 121

string type 124

syntax template 122

using functions 129

value 97

workspace 97, 120

xom.h header file 133

XOM Application Programming Interface 77

xom.h header file 133



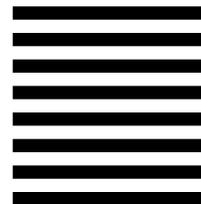
Fold and Tape

Please do not staple

Fold and Tape



NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES



BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

Department G60
International Business Machines Corporation
Information Development
1701 North Street
ENDICOTT NY 13760-5553



Fold and Tape

Please do not staple

Fold and Tape



Program Number: 5694-A01



Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

SC24-5906-00





z/OS DCE

Application Development Guide: Directory Services