



Developing C++ Applications

Solstice Enterprise Manager™

4.1

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303
U.S.A. 650-960-1300

Part No. 806-7970-10
October 2001, Revision A

Copyright 2001 Sun Microsystems, Inc., 901 San Antonio Road, Palo Alto, California 94303 U.S.A. All rights reserved.

This product or document is protected by copyright and distributed under licenses restricting its use, copying, distribution, and decompilation. No part of this product or document may be reproduced in any form by any means without prior written authorization of Sun and its licensors, if any. Third-party software, including font technology, is copyrighted and licensed from Sun suppliers.

Parts of the product may be derived from Berkeley BSD systems, licensed from the University of California. UNIX is a registered trademark in the U.S. and other countries, exclusively licensed through X/Open Company, Ltd.

Sun, Sun Microsystems, the Sun logo, Solstice, Solstice Enterprise Manager, Forte, and Solaris are trademarks, registered trademarks, or service marks of Sun Microsystems, Inc. in the U.S. and other countries. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the U.S. and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

The OPEN LOOK and Sun™ Graphical User Interface was developed by Sun Microsystems, Inc. for its users and licensees. Sun acknowledges the pioneering efforts of Xerox in researching and developing the concept of visual or graphical user interfaces for the computer industry. Sun holds a non-exclusive license from Xerox to the Xerox Graphical User Interface, which license also covers Sun's licensees who implement OPEN LOOK GUIs and otherwise comply with Sun's written license agreements.

RESTRICTED RIGHTS: Use, duplication, or disclosure by the U.S. Government is subject to restrictions of FAR 52.227-14(g)(2)(6/87) and FAR 52.227-19(6/87), or DFAR 252.227-7015(b)(6/95) and DFAR 227.7202-3(a).

DOCUMENTATION IS PROVIDED "AS IS" AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY INVALID.

Copyright 2001 Sun Microsystems, Inc., 901 San Antonio Road, Palo Alto, Californie 94303 Etats-Unis. Tous droits réservés.

Ce produit ou document est protégé par un copyright et distribué avec des licences qui en restreignent l'utilisation, la copie, la distribution, et la décompilation. Aucune partie de ce produit ou document ne peut être reproduite sous aucune forme, par quelque moyen que ce soit, sans l'autorisation préalable et écrite de Sun et de ses bailleurs de licence, s'il y en a. Le logiciel détenu par des tiers, et qui comprend la technologie relative aux polices de caractères, est protégé par un copyright et licencié par des fournisseurs de Sun.

Des parties de ce produit pourront être dérivées des systèmes Berkeley BSD licenciés par l'Université de Californie. UNIX est une marque déposée aux Etats-Unis et dans d'autres pays et licenciée exclusivement par X/Open Company, Ltd.

Sun, Sun Microsystems, le logo Sun, Solstice, Solstice Enterprise Manager, Forte, et Solaris sont des marques de fabrique ou des marques déposées, ou marques de service, de Sun Microsystems, Inc. aux Etats-Unis et dans d'autres pays. Toutes les marques SPARC sont utilisées sous licence et sont des marques de fabrique ou des marques déposées de SPARC International, Inc. aux Etats-Unis et dans d'autres pays. Les produits portant les marques SPARC sont basés sur une architecture développée par Sun Microsystems, Inc.

L'interface d'utilisation graphique OPEN LOOK et Sun™ a été développée par Sun Microsystems, Inc. pour ses utilisateurs et licenciés. Sun reconnaît les efforts de pionniers de Xerox pour la recherche et le développement du concept des interfaces d'utilisation visuelle ou graphique pour l'industrie de l'informatique. Sun détient une licence non exclusive de Xerox sur l'interface d'utilisation graphique Xerox, cette licence couvrant également les licenciés de Sun qui mettent en place l'interface d'utilisation graphique OPEN LOOK et qui en outre se conforment aux licences écrites de Sun.

CETTE PUBLICATION EST FOURNIE "EN L'ETAT" ET AUCUNE GARANTIE, EXPRESSE OU IMPLICITE, N'EST ACCORDEE, Y COMPRIS DES GARANTIES CONCERNANT LA VALEUR MARCHANDE, L'APTITUDE DE LA PUBLICATION A REpondre A UNE UTILISATION PARTICULIERE, OU LE FAIT QU'ELLE NE SOIT PAS CONTREFAISANTE DE PRODUIT DE TIERS. CE DENI DE GARANTIE NE S'APPLIQUERAIT PAS, DANS LA MESURE OU IL SERAIT TENU JURIDIQUEMENT NUL ET NON AVENU.



Please
Recycle



Adobe PostScript

Contents

Preface xxxi

1. Introduction to the Solstice EM C++ Development Environment 1-1

- 1.1 What You Can Develop in the Solstice EM C++ Development Environment 1-1
- 1.2 Solstice EM Network Management Model 1-2
- 1.3 Solstice EM Programming Model 1-2
 - 1.3.1 Solstice EM Application Program Interface (API) Component 1-2
 - 1.3.2 Data Component 1-3
 - 1.3.3 Graphical User Interface (GUI) Component 1-3
- 1.4 Overview of the Application Development Process 1-4
 - 1.4.1 Requirements Analysis and High-Level Design 1-4
 - 1.4.2 Low-Level Design 1-13
 - 1.4.3 Implementation 1-19
 - 1.4.4 Unit Testing and Debugging 1-22
 - 1.4.5 Integration 1-22
 - 1.4.6 System Testing 1-22

2. Modeling Managed Objects 2-1

- 2.1 ISO Management Model 2-1
 - 2.1.1 Managers 2-2

- 2.1.2 Agents 2-3
- 2.1.3 Managed Resources 2-3
- 2.1.4 Managed Objects 2-3
- 2.1.5 Management Protocols 2-3
- 2.1.6 Manager-Agent Hierarchy 2-4
- 2.2 Designing the Object Model 2-5
 - 2.2.1 Defining the Task 2-6
 - 2.2.2 Identifying Managed Object Classes 2-7
 - 2.2.3 Identifying Inheritance Relationships 2-8
 - 2.2.4 Identifying the Characteristics of a Managed Object Class 2-8
 - 2.2.5 Describing the Behavior of Items in the Object Model 2-13
 - 2.2.6 Identifying Containment Relationships 2-14
 - 2.2.7 Grouping Information Into Packages 2-20
 - 2.2.8 Grouping GDMO Definitions Into Documents 2-22
- 2.3 Abstract Syntax Notation #1 (ASN.1) 2-23
 - 2.3.1 Grouping ASN.1 Syntax Definitions Into Modules 2-23
 - 2.3.2 Defining ASN.1 Types 2-24
 - 2.3.3 Defining ASN.1 Values 2-27
 - 2.3.4 Reusing Definitions From Other ASN.1 Modules 2-28
- 2.4 Assigning Unique Identifiers 2-30
 - 2.4.1 Registering an OID 2-30
 - 2.4.2 Guidelines for Allocating Your Own OIDs 2-32
 - 2.4.3 Notation for OIDs 2-33
- 2.5 Obtaining GDMO and ASN.1 Specifications for Objects 2-35
 - 2.5.1 Existing GDMO Definitions 2-35
 - 2.5.2 SNMP MIBs 2-36
- 2.6 Making Your Object Model Available to Solstice EM 2-36
 - 2.6.1 Loading Your Object Model Into the MDR 2-36
 - 2.6.2 Setting Agent Role Behavior of Solstice EM 2-37

3.	Enabling Applications to Access Managed Objects	3-1
3.1	Connecting to an MIS	3-1
3.1.1	Creating and Initializing an Instance of the Platform Class	3-2
3.1.2	Calling the connect Function of the Platform Class	3-3
3.2	Disconnecting From an MIS	3-4
3.3	Bypassing the MIS to Access Solstice EM Databases	3-5
3.3.1	Getting Information Required for a Database Connection	3-5
3.3.2	Passing Database Information to a Database Application Development Tool	3-7
3.3.3	Example Database Connection Program	3-8
4.	Handling Errors	4-1
4.1	Testing for the Success a Function Call	4-1
4.1.1	Using the Overloaded NOT Operator	4-2
4.1.2	Using the get_error_type Function	4-2
4.2	Providing Error Information to Users	4-3
5.	Performing Operations on Managed Objects	5-1
5.1	Management Operations	5-2
5.2	Creating a Managed Object	5-2
5.2.1	Creating and Initializing an Instance of Image	5-3
5.2.2	Activating the Instance of Image	5-4
5.2.3	Verifying if the Managed Object Exists	5-5
5.2.4	Initializing Attributes of the Managed Object	5-5
5.2.5	Adding the Managed Object to the MIS	5-6
5.2.6	Example Object Creation Function	5-7
5.3	Selecting a Managed Object	5-8
5.3.1	Selecting a Managed Object by Specifying its FDN or LDN	5-9
5.3.2	Selecting a Managed Object by Specifying its Nickname	5-10
5.4	Updating an Image Instance	5-14
5.5	Deleting a Managed Object	5-15

- 5.5.1 Removing the Managed Object From the MIS 5-15
 - 5.5.2 Example Object Deletion Function 5-16
- 5.6 Getting Attribute Values From an Object 5-17
- 5.7 Setting Attribute Values of an Object 5-19
 - 5.7.1 Setting Attribute Values in the Image Instance 5-19
 - 5.7.2 Updating the MIS With the Changed Values 5-21
 - 5.7.3 Checking If the MIS Has Been Updated 5-21
 - 5.7.4 Real and Imaginary Values in an Image Instance 5-22
 - 5.7.5 Example 5-23
- 5.8 Performing an Action on an Object 5-23
- 5.9 Tracking Changes to an Object 5-26
 - 5.9.1 Automatically Tracking Changes to an Object 5-26
 - 5.9.2 Manually Tracking Changes to an Object 5-27
- 5.10 Retrieving Data From the Metadata Repository 5-27
 - 5.10.1 Selecting the MDR Managed Object 5-28
 - 5.10.2 Updating the Image Instance That Represents the MDR Managed Object 5-28
 - 5.10.3 Sending the Action Request 5-29
- 5.11 Simulating an Agent Object 5-30
 - 5.11.1 Containing Managed Objects in the Solstice EM MIS 5-31
 - 5.11.2 Making Read-Only Attributes Modifiable 5-31
 - 5.11.3 Loading GDMO Descriptions Into the MIS 5-32
 - 5.11.4 Creating and Modifying Objects in the MIS 5-32
- 5.12 Representing MIS Instances Locally in an Application 5-35

6. Performing Management Operations on Object Collections 6-1

- 6.1 Grouping Managed Objects 6-1
- 6.2 Creating a Container for an Object Collection 6-2
- 6.3 Defining the Membership of an Object Collection 6-3
 - 6.3.1 Defining the Membership by Derivation 6-3
 - 6.3.2 Format of a Derivation String 6-5

6.3.3	Defining the Membership by Enumeration	6-12
6.4	Tracking Changes to an Object Collection	6-13
6.4.1	Maintaining the Membership of an Object Collection	6-13
6.4.2	Setting the Mode of an Object Collection	6-16
6.5	Accessing All Objects in an Object Collection	6-17
6.5.1	Adding All Objects in an Object Collection to the MIS	6-18
6.5.2	Deleting All Objects in an Object Collection	6-18
6.5.3	Setting Attribute Values of All Objects in an Object Collection	6-19
6.5.4	Performing an Action on an All Objects in an Object Collection	6-20
6.5.5	Setting the Synchronization of an Object Collection	6-21
6.6	Accessing Individual Objects in an Object Collection	6-21
6.7	Obtaining All Object Collections for an Object	6-23
7.	Handling Events	7-1
7.1	Event Notifications	7-1
7.2	Processing Information in Event Notifications	7-4
7.2.1	Registering Callback Functions for Event Handling	7-4
7.2.2	Writing Callback Functions for Event Handling	7-6
7.2.3	Controlling Event-Related Updates	7-9
7.3	Scheduling Event Handling	7-11
7.3.1	Scheduling for Applications Without a Graphical User Interface	7-11
7.3.2	Scheduling for Applications With a Graphical User Interface	7-13
7.3.3	Guidelines for Developing Your Own Scheduler	7-15
7.4	Filtering Events	7-16
7.4.1	Selecting Managed Object Classes and Event Types	7-16
7.4.2	Selecting a Subtree of the MIT	7-17
7.4.3	Specifying a Discriminator Construct	7-19
7.5	Simulating an Event	7-20

7.5.1	Simulating an Event Without Using the Solstice EM APIs	7-21
7.5.2	Simulating an Event Programatically	7-21
7.6	Subscribing to Log Record Events	7-23
8.	Performing Asynchronous Management Operations	8-1
8.1	Asynchronous and Synchronous Operation	8-1
8.2	Specifying Asynchronous Operations	8-2
8.2.1	Interactions With the MIS	8-3
8.2.2	Asynchronous Operations on Managed Objects	8-3
8.2.3	Asynchronous Operations on Object Collections	8-4
8.2.4	Asynchronous CMIS Operations on Object Collections	8-5
8.3	Handling Responses From an Asynchronous Operation	8-11
8.3.1	Registering a Callback Function for the Completion of an Asynchronous Operation	8-11
8.3.2	Registering a Callback Function for Handling Responses From Managed Objects	8-12
8.3.3	Writing Callback Functions for Asynchronous Operations	8-13
8.3.4	Scheduling Response Handling	8-20
8.3.5	Adding a Callback to the Scheduler Queue	8-22
8.4	Verifying and Changing the Status of an Asynchronous Operation	8-23
8.4.1	Verifying the Result of an Asynchronous Operation	8-23
8.4.2	Cancelling an Asynchronous Operation	8-24
8.4.3	Changing the Timeout of an Asynchronous Operation	8-25
9.	Encoding and Decoding Complex ASN.1 Values	9-1
9.1	Introduction to the <code>Morf</code> Class	9-1
9.2	Creating Complex ASN.1 Values	9-2
9.2.1	Creating a <code>Morf</code> Instance From String Data	9-2
9.2.2	Creating Simple <code>Morf</code> Instances	9-5
9.2.3	Selecting the Type for a <code>CHOICE</code> Value	9-6
9.2.4	Creating a <code>Morf</code> Instance for ASN.1 ANY Values	9-7

9.2.5	Creating Complex <code>Morf</code> Instances From Other <code>Morf</code> Instances	9-8
9.3	Parsing Complex ASN.1 Values	9-9
9.3.1	Structure of <code>Morf</code> Instances	9-10
9.3.2	Overview of Functions for Parsing <code>Morf</code> Instances	9-11
9.3.3	Parsing <code>CHOICE</code> Values	9-12
9.3.4	Parsing List Values	9-13
9.3.5	Getting Objects Associated With a <code>Morf</code> Instance	9-15
9.3.6	Getting Metainformation About the ASN.1 Type of a <code>Morf</code> Instance	9-16
9.3.7	Example of Parsing a <code>Morf</code> Instance	9-23
9.4	Decoding Complex ASN.1 Values	9-24
9.4.1	Getting a String Representation of a <code>Morf</code> Instance	9-25
9.4.2	Extracting a Value in a <code>Morf</code> Instance as a New <code>Morf</code> Instance	9-27
9.4.3	Getting the Value Assigned to a <code>Morf</code> Instance	9-28
9.4.4	Getting Scalar Values Assigned to a <code>Morf</code> Instance	9-29
9.5	Using the <code>MorfBuilder</code> Class	9-32
9.5.1	Constructing a <code>MorfBuilder</code> Instance	9-33
9.5.2	Adding Data to a <code>MorfBuilder</code> Instance	9-33
9.5.3	Selecting a Syntax for <code>CHOICE</code> Values	9-35
9.5.4	Setting a Navigation Type for <code>SEQUENCE</code> Values	9-37
9.5.5	Validating the Data in a <code>MorfBuilder</code> Instance	9-38
9.5.6	Assembling <code>MorfBuilder</code> Data Into a Single <code>Morf</code> Instance	9-39
10.	Developing Object Behaviors	10-1
10.1	ODT Overview	10-2
10.1.1	Supporting Functions	10-2
10.1.2	Object Development Components	10-3
10.2	Object Interfaces	10-3
10.2.1	Object Behavior Interface	10-4
10.2.2	Object Services API	10-4

10.3	Object Development Overview	10-5
10.3.1	Possible Errors	10-6
10.3.2	Sanity Check Procedure	10-6
10.4	Object Code Generator Utility	10-8
10.4.1	Generated Code Interfaces	10-9
10.4.2	Code Generation Components	10-10
10.4.3	Using the Object Code Generator Utility	10-12
10.4.4	Configuring the Object Code Generator Utility	10-13
10.4.5	How Filter Attributes Affect Code Generation	10-13
10.5	Implementing GDMO Specified Object Behavior	10-14
10.5.1	MIS Object Modeling Concepts	10-14
10.5.2	Asynchronous Interface Behavior	10-15
10.5.3	Sub Operations: subfetch, subread, subwrite, substore	10-21
10.5.4	Propagation of Errors	10-22
10.5.5	Serialization of Object Requests	10-25
10.6	Debugging Objects	10-26
10.6.1	Process	10-26
10.6.2	Dynamic Loading in Solstice EM	10-27
10.6.3	ASN.1 and GDMO Debugging	10-28
10.6.4	Printing ASN.1 Values in Human-Readable Form	10-28
10.6.5	Debugging Flags	10-29
10.7	Generated Files	10-29
10.7.1	Makefile (<i>Makefile.className</i>)	10-30
10.7.2	Readme File (<i>README.className</i>)	10-30
10.7.3	User Header File (<i>className_user.odt.hh</i>)	10-30
10.7.4	PMI Client Create Program for Object Instantiation (<i>pmi.className.cc</i>)	10-31
10.7.5	User Code File (<i>className_user.odt.cc</i>)	10-31
10.7.6	Dynamic Loading File (<i>className.load</i>)	10-32
10.7.7	Dynamic Unloading File (<i>className.unload</i>)	10-32

10.8	TRY Exception Macros	10-32
10.8.1	Overview	10-32
10.8.2	Code Structure	10-33
10.8.3	Code Examples	10-33
10.9	Object Development Examples	10-34
10.9.1	Compiling All Examples	10-34
10.9.2	cellSample	10-35
10.9.3	demoPing	10-36
10.9.4	demoregistry	10-42
10.9.5	demoServer	10-47
10.9.6	diskInfo	10-48
10.10	Object Development Scenario Using Chai Object	10-49
10.10.1	Creating Your Own Object Class	10-49
10.10.2	Debugging Flags	10-52
10.10.3	Sample Behavior Implementation	10-52
10.10.4	chai Object Class Definitions	10-54
10.10.5	Sample PMI Program to Create a New chai Object Instance	10-58
10.10.6	Example Generated Code in .cc File	10-62
10.10.7	Example Generated Code in .hh File	10-82
11.	Writing Management Protocol Adaptors (MPAs)	11-1
11.1	Review of MIS Architecture	11-2
11.2	Initializing Management Protocol Adaptors and Protocol Driver Modules	11-4
11.2.1	Services Access Points (SAPs)	11-4
11.2.2	Initializing a Management Protocol Adaptor	11-6
11.2.3	Initializing a Protocol Driver Module	11-9
11.3	Routing Messages	11-12
11.3.1	How Messages are Routed to the Adaptors	11-12
11.3.2	MPA and PDM Addresses	11-14
11.3.3	FDN Table Configuration Options	11-17

11.3.4	Source and Destination Fields in the Message	11-18
11.4	MPA/PDM Request Management	11-19
11.4.1	Asynchronous Request Code Specifics	11-20
11.4.2	Validating Requests	11-22
11.4.3	Matching Requests to Responses	11-22
11.5	Timer Management	11-23
11.5.1	Timer Management Interface	11-23
11.5.2	Stopping a Timer	11-26
11.6	File Descriptor Management	11-26
11.6.1	Asynchronous File I/O	11-26
11.6.2	Example of a Read Callback Implementation	11-27
11.7	Notifications	11-30
11.7.1	Creating a Notification	11-31
11.8	Sample MPA/PDM Source Code	11-33
11.8.1	Files and Configuration	11-33
11.9	Developing an Adaptor	11-35
11.9.1	Defining the Management Information Model	11-35
11.9.2	The Request Management Interface	11-36
11.9.3	The Protocol Code	11-36
12.	Controlling Access to Applications and Data	12-1
12.1	Access Control Levels	12-1
12.1.1	Application-Level Access Control	12-2
12.1.2	Application-Feature-Level Access Control	12-2
12.1.3	Managed-Object-Level Access Control	12-3
12.1.4	Event Notification Access Control	12-3
12.1.5	Management Protocol Adapter (MPA) Access Control	12-3
12.2	Enforcing Predefined Access Control Rules	12-4
12.2.1	Defining Access Control Rules	12-4
12.2.2	Enforcing Application-Level and Application-Feature-Level Access Control	12-6

12.2.3	Handling Denial of Access to Managed Objects	12-8
12.3	Modifying Access Control Information	12-8
12.3.1	Activating Access Control for the Solstice EM Platform	12-9
12.3.2	Adding a User to a Privilege Group	12-10
12.3.3	Listing All Application Features Under Access Control	12-16
12.3.4	Adding Applications and Application Features to a Privilege Group	12-18
12.3.5	Defining a Target	12-18
12.3.6	Defining a Security Rule	12-26
12.3.7	Handling Access Control Errors	12-31
12.4	Getting Access Control Defaults	12-32
12.4.1	Getting the Default Enforcement Action for All Management Operations	12-33
12.4.2	Getting the Default Enforcement Action for All Events	12-33
12.4.3	Getting a List of Trusted Hosts	12-34
12.4.4	Getting the Access Control Denial Granularity	12-35
12.4.5	Getting the Access Control Domain	12-36
12.5	Keeping Event Notifications Private	12-36
12.5.1	Assigning an Owner to a Log	12-36
12.5.2	Enabling Access Control for the Log Server	12-39
12.6	Making MPAs Secure	12-40
12.6.1	Subscribing to Access Control Events	12-40
12.6.2	Creating and Initializing an Instance of the ACE Class	12-42
12.6.3	Processing Information in Access Control Events	12-43
12.6.4	Implementing a Class Derived From AuxServerUtils	12-44
12.6.5	Calling Access Control Decision and Enforcement Functions	12-44
13.	Optimizing Performance	13-1
13.1	General Guidelines for Optimizing Performance	13-1
13.2	Selectively Activating Image Instances	13-2

13.3	Filtering Events	13-3
13.4	Writing Your Own Classes to Represent Managed Objects	13-3
13.5	Using the Low-Level PMI	13-3
14.	Guidelines for Compiling and Linking Applications	14-1
14.1	Compiler Version Requirements	14-1
14.2	Header Files and Libraries	14-1
14.3	Options for Locating Header Files and Libraries	14-7
14.4	Compilation Flags	14-8
15.	Troubleshooting	15-1
15.1	Testing and Debugging Programs	15-1
15.1.1	Verifying GDMO and ASN.1 Syntax and Logic	15-2
15.1.2	Trapping Errors in PMI Function Calls	15-2
15.1.3	Trapping Programming Logic Errors	15-2
15.1.4	Monitoring Protocol Translation by an MPA	15-3
15.1.5	Reloading GDMO Documents	15-4
15.2	Monitoring Communications With the MIS	15-7
15.2.1	Starting <code>em_debug</code>	15-7
15.2.2	Interpreting <code>em_debug</code> Messages	15-8
15.2.3	Full List of <code>em_debug</code> Message Types	15-15
15.3	Avoiding Common Problems	15-19
15.3.1	Verifying Attribute and Class Names	15-19
15.3.2	Creating Automatically Named Managed Object Instances Appropriately	15-20
15.3.3	Testing That Scopes and Filters are Supported	15-20
15.4	Example Troubleshooting Scenarios	15-23
15.4.1	Failure to Set an Attribute Value	15-23
15.4.2	Failure to Process Notifications	15-25
16.	Integrating Applications With Solstice EM	16-1

16.1	Adding an Application to a Tools Window	16-1
16.2	Extending the Tools Menu of a Solstice EM Tool	16-4
16.3	Customizing the Network Views Tool	16-6
16.3.1	Extending the Actions Menu of the Network Views Tool	16-6
16.3.2	Setting the Activation of a Topology Type	16-9
17.	Writing RPC Agents for Solstice EM	17-1
17.1	Manager-Agent Model	17-1
17.2	Types of Agents	17-2
17.3	Steps for Writing an Agent	17-3
17.4	Solstice EM Integration	17-4
17.4.1	Installing the Agent	17-4
17.4.2	Updating the Solstice EM MIS Database	17-5
A.	Solstice EM C++ Source Code Examples	A-1
A.1	Guidelines for Compiling the Examples	A-1
A.2	Satellite Example	A-2
A.3	High-Level PMI Examples	A-3
A.3.1	Managed Object Examples	A-3
A.3.2	Object Collection Examples	A-5
A.3.3	Event Handling Examples	A-5
A.3.4	Log Record Handling Examples	A-6
A.3.5	Network Topology Examples	A-7
A.3.6	FDN Translation Examples	A-7
A.3.7	Graphical Application Examples	A-7
A.3.8	MDR Action Examples	A-8
A.3.9	Encoding and Decoding Examples	A-8
A.4	Scenario Examples	A-8
A.5	Security Examples	A-9
A.5.1	Access Control API Examples	A-9
A.5.2	Access Control Engine API Examples	A-9

A.5.3	Password Request Example	A-10
A.5.4	Application-Feature-Level Example	A-10
A.6	Low-Level PMI Examples	A-10
A.7	Object Modeling Example	A-11
A.8	Object Development Examples	A-11
A.9	Miscellaneous Examples	A-12
B.	Standards Reference and Further Reading	B-1
B.1	Standards Reference	B-1
B.2	Terminology References	B-3
B.3	Further Reading	B-5
C.	GDMO Templates	C-1
C.1	Conventions Used in the Definitions	C-1
C.2	Managed Object Class Template	C-2
C.2.1	Managed Object Class Template Format	C-2
C.2.2	Managed Object Class Template Constructs	C-3
C.3	Name Binding Template	C-4
C.3.1	Name Binding Template Format	C-4
C.3.2	Name Binding Template Constructs	C-5
C.4	Package Template	C-7
C.4.1	Package Template Format	C-7
C.4.2	Package Template Constructs	C-8
C.4.3	PropertyList Supporting Production	C-9
C.5	Attribute Template	C-11
C.5.1	Attribute Template Format	C-12
C.5.2	Attribute Template Constructs	C-12
C.6	Action Template	C-13
C.6.1	Action Template Format	C-14
C.6.2	Action Template Constructs	C-14
C.7	Notification Template	C-15

C.7.1	Notification Template Format	C-16
C.7.2	Notification Template Constructs	C-16
C.8	Parameter Template	C-18
C.8.1	Parameter Template Format	C-18
C.8.2	Parameter Template Constructs	C-19
C.9	Attribute Group Template	C-20
C.9.1	Attribute Group Template Format	C-20
C.9.2	Attribute Group Template Constructs	C-21
C.10	Behaviour Template	C-22
C.10.1	Behaviour Template Format	C-22
C.10.2	Behaviour Template Constructs	C-22

Index **Index-1**

Tables

TABLE 2-1	Attributes for the <code>satellite</code> Managed Object Class	2-9
TABLE 2-2	Attributes for the <code>channel</code> Managed Object Class	2-10
TABLE 2-3	Attributes for the <code>dish</code> Managed Object Class	2-10
TABLE 2-4	Notifications for the Satellite Example	2-12
TABLE 2-5	ASN.1 Universal Types	2-26
TABLE 3-1	Functions for Getting Information About a Database	3-7
TABLE 5-1	Functions for Getting Attribute Values From an <code>Image</code> Instance	5-17
TABLE 5-2	Functions for Setting Attribute Values in an <code>Image</code> Instance	5-19
TABLE 5-3	Operations for Setting Attributes	5-21
TABLE 5-4	Functions for Getting the Value Last Set by an Application	5-22
TABLE 5-5	Functions for Sending an Action Request to a Managed Object	5-24
TABLE 5-6	Actions for Retrieving Metadata From the MDR	5-29
TABLE 5-7	Variable Parts of the Format of an <code>em_objop</code> Script	5-33
TABLE 6-1	Scope Values in a Derivation String	6-6
TABLE 6-2	Filter Operator Keywords	6-9
TABLE 6-3	Comparison Keywords in a Filter Without Substrings	6-10
TABLE 6-4	Part Keywords in a Substring	6-11
TABLE 6-5	Functions for Setting Attribute Values in an Object Collection	6-19
TABLE 7-1	Event Types Defined in Recommendation ITU-T X.721/ISO-10165-2	7-3

TABLE 7-2	Event Types Recognized by the <code>when</code> Function	7-5
TABLE 7-3	Functions for Extracting Information from Event Notifications	7-7
TABLE 7-4	Scope Values in a Subtree for the <code>replace_discriminator</code> Function	7-18
TABLE 8-1	Synchronous and Asynchronous Functions of the <code>Platform</code> Class	8-3
TABLE 8-2	Synchronous and Asynchronous Functions of the <code>Image</code> Class	8-3
TABLE 8-3	Synchronous and Asynchronous Functions of the <code>Album</code> Class	8-4
TABLE 8-4	CMIS Operations Supported by the <code>Album</code> Class	8-5
TABLE 8-5	Functions of the <code>Album</code> Class for Requesting CMIS Operations	8-7
TABLE 8-6	Operations for the <code>set_operator</code> Function	8-9
TABLE 8-7	Information Available From All Responses	8-17
TABLE 8-8	Information Available Only From Action Replies	8-17
TABLE 9-1	Functions for Assigning Scalar Values to a <code>Morf</code> Instance	9-5
TABLE 9-2	Functions for Parsing <code>Morf</code> Instances	9-11
TABLE 9-3	Functions of the <code>Asn1Type</code> Class For Parsing <code>Morf</code> Instances	9-12
TABLE 9-4	Functions for Retrieving Information About the Type Instance	9-15
TABLE 9-5	Default String Representation of Values by Type in a <code>Morf</code> Instance	9-25
TABLE 9-6	Identifiers for Format Bits Arguments	9-27
TABLE 9-7	Functions for Extracting Numeric Scalars Into Numeric Types	9-29
TABLE 9-8	Functions of the <code>Asn1Value</code> Class For Decoding Data	9-30
TABLE 9-9	Constructors of the <code>MorfBuilder</code> Class	9-33
TABLE 10-1	OCG Command Line Options	10-12
TABLE 10-2	Object Development Tool Configuration File Parameters	10-13
TABLE 10-3	Behavior Abstractions	10-14
TABLE 10-4	Interfaces for CMIS Requests	10-16
TABLE 10-5	Order of CMIS Request Interfaces	10-16
TABLE 10-6	Attribute Class Helper Methods	10-30
TABLE 10-7	Action Class Helper Methods	10-31
TABLE 10-8	Object Development Examples	10-34

TABLE 11-1	Message Services	11-5
TABLE 11-2	MIS and MPA/PDM Connections	11-20
TABLE 11-3	MPA Example Files	11-33
TABLE 12-1	Predefined Privilege Groups	12-10
TABLE 12-2	Predefined Targets	12-19
TABLE 12-3	Operations for a Target	12-21
TABLE 12-4	Scope Values in the Constructor of <code>ACScope</code>	12-24
TABLE 12-5	Predefined Security Rules	12-26
TABLE 12-6	Enforcement Actions	12-30
TABLE 12-7	Access Control Denial Granularity Levels	12-35
TABLE 14-1	Header Files and Libraries for the Solstice EM Schedulers	14-2
TABLE 14-2	Header Files and Libraries for the Solstice EM API Classes	14-2
TABLE 14-3	Compilation Flags for Applications Developed With the Solstice EM C++ APIs	14-8
TABLE 15-1	Commonly Used <code>em_debug</code> Message Types	15-8
TABLE 15-2	ASN.1 Data Types and Tag Numbers	15-10
TABLE 15-3	<code>em_debug</code> Message Types	15-15
TABLE 16-1	Configuration Files for Solstice EM Tools Windows	16-2
TABLE 16-2	Variable Parts in a Configuration File Entry for a Tools Window	16-2
TABLE 16-3	Configuration Files for Solstice EM Tools	16-4
TABLE 16-4	Variable Parts in a Configuration File Entry for a Solstice EM Tool	16-5
TABLE 16-5	Variable Parts in a Configuration File Entry for the Actions Menu	16-7
TABLE 16-6	Variable Parts of the Configuration File Entry That Sets Activations	16-9
TABLE A-1	Subdirectories of the Satellite Example Directory	A-2
TABLE A-2	Managed Object Examples for the High-Level PMI	A-3
TABLE A-3	Object Collection Examples for the High-Level PMI	A-5
TABLE A-4	Event Handling Examples for the High-Level PMI	A-5
TABLE A-5	Log Record Handling Examples for the High-Level PMI	A-6
TABLE A-6	Network Topology Examples for the High-Level PMI	A-7
TABLE A-7	FDN Translation Examples for the High-Level PMI	A-7

TABLE A-8	Scenario Examples	A-8
TABLE A-9	Access Control API Examples	A-9
TABLE 17-1	Access Control Engine API Examples	A-9
TABLE A-10	Low-Level PMI Example Programs	A-10
TABLE A-11	Object Development Examples	A-11
TABLE A-12	Miscellaneous API Examples	A-12
TABLE B-1	ISO Specifications for Terminology Definitions	B-3
TABLE C-1	Managed Object Class Template Constructs	C-3
TABLE C-2	Name Binding Template Constructs	C-5
TABLE C-3	Package Template Constructs	C-8
TABLE C-4	<code>propertyList</code> Supporting Production Definitions	C-10
TABLE C-5	Attribute Template Constructs	C-12
TABLE C-6	Action Template Constructs	C-14
TABLE C-7	Notification Template Constructs	C-16
TABLE C-8	Parameter Template Constructs	C-19
TABLE C-9	Attribute Group Template Constructs	C-21
TABLE C-10	Behaviour Template Constructs	C-22

Code Samples

CODE EXAMPLE 2-1	GDMO Definition of the <code>dish</code> Managed Object Class	2-7
CODE EXAMPLE 2-2	GDMO Definition of the <code>censureButton</code> Attribute	2-11
CODE EXAMPLE 2-3	ASN.1 Syntax Definition of the <code>ButtonPress</code> Data Type	2-11
CODE EXAMPLE 2-4	GDMO Specification of the <code>objectCreation</code> Event	2-12
CODE EXAMPLE 2-5	Behavior of the <code>packetRetries</code> Attribute	2-13
CODE EXAMPLE 2-6	GDMO Definition of the <code>satellite-system</code> Name Binding	2-16
CODE EXAMPLE 2-7	GDMO Definition of the Example MIT	2-17
CODE EXAMPLE 2-8	GDMO Definition of the <code>dishPackage</code> Package	2-21
CODE EXAMPLE 2-9	Naming a GDMO Document	2-22
CODE EXAMPLE 2-10	Beginning and End of an ASN.1 Module	2-24
CODE EXAMPLE 2-11	Definition of the <code>CurrentLogSize</code> ASN.1 Type	2-25
CODE EXAMPLE 2-12	Definition of the <code>SatelliteData</code> ASN.1 Type	2-25
CODE EXAMPLE 2-13	Definition of the <code>SatelliteSeq</code> ASN.1 Type	2-25
CODE EXAMPLE 2-14	Specifying a Range of Allowed Values for an ASN.1 Type	2-27
CODE EXAMPLE 2-15	Defining an ASN.1 Value	2-27
CODE EXAMPLE 2-16	Importing ASN.1 Definitions	2-29
CODE EXAMPLE 2-17	Exporting ASN.1 Definitions	2-29
CODE EXAMPLE 2-18	OIDs for Branches of the Subtree in the Satellite Example	2-32
CODE EXAMPLE 2-19	OIDs for the Satellite Example	2-32

CODE EXAMPLE 2-20	Labelling an <code>OID</code>	2-34
CODE EXAMPLE 2-21	Using an <code>OID</code> Label in an <code>OID</code> Assignment	2-34
CODE EXAMPLE 3-1	Creating and Initializing a <code>Platform</code> Instance	3-2
CODE EXAMPLE 3-2	Calling the <code>connect</code> Function	3-3
CODE EXAMPLE 3-3	Calling the <code>disconnect</code> Function	3-4
CODE EXAMPLE 3-4	Creating and Initializing an Instance of the <code>EMDBConnectInfo</code> Class	3-6
CODE EXAMPLE 3-5	Connecting to a Solstice EM Database	3-8
CODE EXAMPLE 3-6	Getting Information Directly From a Solstice EM Database	3-8
CODE EXAMPLE 4-1	Using the Overloaded <code>NOT</code> Operator for Error Checking	4-2
CODE EXAMPLE 4-2	Using the <code>get_error_type</code> Function for Error Checking	4-3
CODE EXAMPLE 5-1	Creating and Initializing an <code>Image</code> Instance	5-3
CODE EXAMPLE 5-2	Activating an <code>Image</code> Instance	5-4
CODE EXAMPLE 5-3	Verifying if a Managed Object Exists	5-5
CODE EXAMPLE 5-4	Initializing Managed Object Attributes	5-6
CODE EXAMPLE 5-5	Adding a Managed Object to an <code>MIS</code>	5-7
CODE EXAMPLE 5-6	Example Object Creation Function	5-7
CODE EXAMPLE 5-7	Selecting a Managed Object by Specifying its <code>FDN</code>	5-9
CODE EXAMPLE 5-8	Mappings Between <code>FDNs</code> and Nicknames	5-12
CODE EXAMPLE 5-9	Getting the <code>Image</code> Instance Associated With a Nickname	5-13
CODE EXAMPLE 5-10	Updating an <code>Image</code> Instance	5-14
CODE EXAMPLE 5-11	Removing a Managed Object from the <code>MIS</code>	5-15
CODE EXAMPLE 5-12	Example Object Deletion Function	5-16
CODE EXAMPLE 5-13	Getting Attribute Values	5-18
CODE EXAMPLE 5-14	Setting an Attribute Value	5-23
CODE EXAMPLE 5-15	Sending an Action Request	5-24
CODE EXAMPLE 5-16	GDMO Specification of the <code>topoNodeGetByName</code> Action	5-25
CODE EXAMPLE 5-17	ASN.1 Definitions of Data Types Used by <code>topoNodeGetByName</code>	5-25
CODE EXAMPLE 5-18	Setting the <code>TACKMODE</code> Property of an <code>Image</code> Instance	5-26

CODE EXAMPLE 5-19 `Selecting the MDR Managed Object` 5-28

CODE EXAMPLE 5-20 `Updating the Image Instance that Represents the MDR` 5-29

CODE EXAMPLE 5-21 `Sending an Action Request` 5-30

CODE EXAMPLE 5-22 `Name Binding Clause for Instantiation Under system` 5-31

CODE EXAMPLE 5-23 `em_objop Script for Creating an Object` 5-34

CODE EXAMPLE 5-24 `em_objop Script for Setting an Attribute Value` 5-34

CODE EXAMPLE 5-25 `em_objop Script for Deleting an Object` 5-35

CODE EXAMPLE 5-26 `em_objop Script for Deriving an Album Instance` 5-35

CODE EXAMPLE 5-27 `C++ Representation of Managed Object Classes` 5-36

CODE EXAMPLE 6-1 `Creating and Initializing an Album Instance` 6-2

CODE EXAMPLE 6-2 `Setting a Derivation String` 6-4

CODE EXAMPLE 6-3 `Starting a Derivation` 6-5

CODE EXAMPLE 6-4 `Selecting All log Objects` 6-11

CODE EXAMPLE 6-5 `Selecting All Enabled Instances of log` 6-11

CODE EXAMPLE 6-6 `Selecting all Objects That are not Instances of log` 6-12

CODE EXAMPLE 6-7 `Equivalent Derivation Strings` 6-12

CODE EXAMPLE 6-8 `Setting the TACKMODE Property of an Album Instance` 6-14

CODE EXAMPLE 6-9 `Using Callback Functions With an Object Collection` 6-15

CODE EXAMPLE 6-10 `Setting the Mode of an Object Collection` 6-17

CODE EXAMPLE 6-11 `Retrieving Objects From an Object Collection` 6-22

CODE EXAMPLE 6-12 `Obtaining all Object Collections for an Object` 6-23

CODE EXAMPLE 7-1 `Specification of Event Types Supported by a Managed Object Class` 7-2

CODE EXAMPLE 7-2 `GDMO Specification of the objectCreation Event` 7-2

CODE EXAMPLE 7-3 `Registering Callback Functions` 7-6

CODE EXAMPLE 7-4 `Callback Function` 7-8

CODE EXAMPLE 7-5 `Tracking Changes From Within a Callback` 7-10

CODE EXAMPLE 7-6 `Calling the dispatch_recursive Function` 7-12

CODE EXAMPLE 7-7 `Calling the dispatch_main_loop Function` 7-12

CODE EXAMPLE 7-8	Contents of the <code>dispatch_main_loop</code> Function	7-13
CODE EXAMPLE 7-9	Initializing and Activating the <code>xtsched</code> Scheduler	7-14
CODE EXAMPLE 7-10	Disabling and Enabling the Processing of X Events	7-15
CODE EXAMPLE 7-11	Selecting Managed Object Classes and Event Types	7-16
CODE EXAMPLE 7-12	Selecting a Subtree of the MIT	7-18
CODE EXAMPLE 7-13	Specifying a Discriminator Construct	7-20
CODE EXAMPLE 7-14	Simulating an Event Programmatically	7-22
CODE EXAMPLE 7-15	Subscribing to Log Record Events	7-23
CODE EXAMPLE 8-1	Selecting Managed Objects for a CMIS Operation	8-6
CODE EXAMPLE 8-2	Requesting an Asynchronous CMIS <code>M-SET</code> Operation	8-9
CODE EXAMPLE 8-3	Registering a Callback for an Asynchronous Operation	8-12
CODE EXAMPLE 8-4	Registering a Callback Function for Response Handling	8-13
CODE EXAMPLE 8-5	Callback for Completion of an Asynchronous Operation	8-14
CODE EXAMPLE 8-6	Correct Use of Data Passed by the Scheduler	8-16
CODE EXAMPLE 8-7	Incorrect Use of Data Passed by the Scheduler	8-16
CODE EXAMPLE 8-8	Callback for Handling Responses From Managed Objects	8-18
CODE EXAMPLE 8-9	Scheduling Nonblocking Asynchronous Response Handling	8-21
CODE EXAMPLE 8-10	Scheduling Blocking Asynchronous Response Handling	8-22
CODE EXAMPLE 8-11	Verifying the Result of an Asynchronous Operation	8-24
CODE EXAMPLE 8-12	Changing the Timeout of an Asynchronous Operation	8-25
CODE EXAMPLE 9-1	ASN.1 Syntax of <code>DestructSet</code>	9-3
CODE EXAMPLE 9-2	Constructing a Morf From a String	9-4
CODE EXAMPLE 9-3	Selecting the Type for a <code>CHOICE</code> Value	9-6
CODE EXAMPLE 9-4	ASN.1 Syntax of <code>AttributeValueAssertion</code>	9-7
CODE EXAMPLE 9-5	Assigning a Value to an Instance of the ASN.1 <code>ANY</code> Type	9-7
CODE EXAMPLE 9-6	Extracting Data From a <code>CHOICE</code> Value	9-12
CODE EXAMPLE 9-7	Using a Queue to Parse a List	9-14
CODE EXAMPLE 9-8	Obtaining <code>BIT</code> <code>STRING</code> and <code>ENUMERATED</code> Identifiers	9-17

CODE EXAMPLE 9-9 Obtaining the Range Limits for a Value 9-19

CODE EXAMPLE 9-10 Obtaining the Size Constraints of a Value 9-22

CODE EXAMPLE 9-11 Sample Function for Parsing a `Morf` Instance 9-23

CODE EXAMPLE 9-12 ASN.1 Type Definition of the `GeoLocation` Type 9-25

CODE EXAMPLE 9-13 Default String Representation of a `GeoLocation` Value 9-26

CODE EXAMPLE 9-14 Using Navigation Strings With the `extract` Function 9-28

CODE EXAMPLE 9-15 Decoding a `Morf` Instance Directly Into an `Oid` Instance 9-31

CODE EXAMPLE 9-16 Using `set` to Update a `MorfBuilder` Instance 9-34

CODE EXAMPLE 9-17 Selecting a Syntax For a `CHOICE` Value 9-36

CODE EXAMPLE 9-18 Using `get_prop` and `set_prop` 9-37

CODE EXAMPLE 12-1 `em_accesscmd` Script 12-5

CODE EXAMPLE 12-2 Controlling Application- and Application-Feature-Level Access 12-7

CODE EXAMPLE 12-3 Activating Access Control for the Solstice EM Platform 12-9

CODE EXAMPLE 12-4 Creating a Privilege Group 12-12

CODE EXAMPLE 12-5 Creating a User 12-12

CODE EXAMPLE 12-6 Creating an Access Control List 12-13

CODE EXAMPLE 12-7 Adding a User and Storing an Access Control List 12-14

CODE EXAMPLE 12-8 Adding a User to a Privilege Group 12-15

CODE EXAMPLE 12-9 Listing Applications Under Application-Feature-Level Access Control 12-16

CODE EXAMPLE 12-10 Listing Application Features Under Access Control 12-17

CODE EXAMPLE 12-11 Creating a Target 12-20

CODE EXAMPLE 12-12 Defining the List of Operations for a Target 12-22

CODE EXAMPLE 12-13 Storing a Target Persistently in the MIS 12-25

CODE EXAMPLE 12-14 Creating a Security Rule 12-28

CODE EXAMPLE 12-15 Adding a Privilege Group to a Security Rule 12-28

CODE EXAMPLE 12-16 Adding a Target to a Security Rule 12-29

CODE EXAMPLE 12-17 Defining the Enforcement Action of a Security Rule 12-30

CODE EXAMPLE 12-18 Storing a Security Rule 12-31

CODE EXAMPLE 12-19	Error Handling Example	12-32
CODE EXAMPLE 12-20	Getting Default Access Control for All Operations	12-33
CODE EXAMPLE 12-21	Getting the Default Enforcement Action for All Events	12-34
CODE EXAMPLE 12-22	Getting a List of Trusted Hosts	12-34
CODE EXAMPLE 12-23	Getting the Access Control Denial Granularity	12-35
CODE EXAMPLE 12-24	Getting the Access Control Domain	12-36
CODE EXAMPLE 12-25	Assigning an Owner to a Log	12-38
CODE EXAMPLE 12-26	Subscribing to Access Control Events	12-41
CODE EXAMPLE 12-27	Creating and Initializing an Instance of the <code>ACE</code> Class	12-42
CODE EXAMPLE 12-28	Registering a Callback for Access Control Events	12-44
CODE EXAMPLE 13-1	Selectively Activating an <code>Image</code> Instance	13-2
CODE EXAMPLE 13-2	Getting Information From an Object Collection	13-4
CODE EXAMPLE 13-3	Callback for Handling Responses to a <code>Get</code> Request	13-5
CODE EXAMPLE 15-1	Sample <code>em_debug</code> Output	15-11
CODE EXAMPLE 15-2	Replacing a Scope and a Filter With Multiple Derivations	15-21
CODE EXAMPLE 16-1	Network Tools Window Configuration File Entry	16-3
CODE EXAMPLE 16-2	Configuration File Entry for Extending the Tools Menu	16-5
CODE EXAMPLE 16-3	Extending the Actions Menu of the Network Views Tool	16-8
CODE EXAMPLE 16-4	Setting the Activations of Topology Types	16-10

Figures

- FIGURE 1-1 Architecture of the Solstice EM C++ APIs 1-14
- FIGURE 2-1 ISO Network Management Model 2-2
- FIGURE 2-2 Manager-Agent Hierarchy 2-4
- FIGURE 2-3 Example Inheritance Tree 2-8
- FIGURE 2-4 Example MIT 2-14
- FIGURE 2-5 Containment Tree and Object Naming 2-19
- FIGURE 2-6 ISO Registration Tree 2-31
- FIGURE 6-1 Scope Values 6-7
- FIGURE 6-2 Combination of a Scope and a Filter 6-8
- FIGURE 10-1 ODT components 10-3
- FIGURE 10-2 ODT Framework, with Generated Code Interface Highlighted 10-9
- FIGURE 10-3 Code Generation Components 10-10
- FIGURE 10-4 Sequence Diagram for `M_GET` operation 10-18
- FIGURE 10-5 Sequence Diagram for `M_ACTION` 10-19
- FIGURE 10-6 Sequence Diagram for `M_SET` 10-20
- FIGURE 11-1 MIS Architecture 11-3
- FIGURE 11-2 Potential Real World Configuration 11-19

Preface

Developing C++ Applications explains how to use the C++ APIs of Solstice™ Enterprise Manager™ (Solstice EM) to develop network management applications. Use this book with the *C++ API Reference*.

Who Should Use This Book

This book is intended for software developers who are using Solstice EM to develop network management applications. Knowledge of C++, object-oriented design, and object-oriented programming are assumed.

This book assumes that you are familiar with the principles and concepts of network management and that you have had some experience managing or developing applications to manage a network.

Before You Read This Book

If you have just acquired the Solstice EM product, read the *Customizing Guide* for an overview of the Solstice EM product functions, features, and components. Read the *Release Notes* for information on installing and starting the product, compatibility issues, minimum hardware and software requirements, known problems, an inventory of the product components, and late breaking information.

How This Book Is Organized

This book is organized as follows:

Chapter 1 “Introduction to the Solstice EM C++ Development Environment” introduces the C++ APIs of Solstice EM. This chapter explains which types of applications you can develop with the C++ APIs of Solstice EM. This chapter also introduces the Solstice EM network management and programming models, and provides an overview of the application development process.

Chapter 2 “Modeling Managed Objects” explains how to write an object model by using Guidelines for the Definition of Managed Objects (GDMO) and Abstract Syntax Notation One (ASN.1). This chapter introduces the ISO management model on which Solstice EM is based. In addition, this chapter provides detailed instructions on how to fill out GDMO templates and write ASN.1 module specifications. This chapter also explains how to make your object model available to Solstice EM.

Chapter 3 “Enabling Applications to Access Managed Objects” explains how to connect an application to the management information server (MIS) to enable the application to access managed objects. This chapter also explains how to bypass the MIS to access Solstice EM databases directly.

Chapter 4 “Handling Errors” explains how to handle errors in function calls by using the operators and functions that Solstice EM provides for error handling.

Chapter 5 “Performing Operations on Managed Objects” explains how to perform management operations on individual managed objects to control and monitor managed resources in a network. This chapter also explains how to simulate an agent object.

Chapter 6 “Performing Management Operations on Object Collections” explains how to perform bulk operations on managed objects by selecting multiple managed objects to be the subject of a management operation.

Chapter 7 “Handling Events” explains how to enable an application to receive event notifications that are emitted by managed objects and process the information contained in event notifications.

Chapter 8 “Performing Asynchronous Management Operations” explains how to perform asynchronous operations on managed objects and object collections. This chapter also explains how to handle responses to asynchronous operations that an application receives.

Chapter 9 “Encoding and Decoding Complex ASN.1 Values” explains how to enable an application to encode complex ASN.1 values for transmission in management requests. This chapter also explains how to enable an application to decode complex ASN.1 values received in responses and event notifications.

Chapter 10 “Developing Object Behaviors” explains how to use the Solstice EM object development tools (ODT) to develop object behaviors.

Chapter 11 “Writing Management Protocol Adaptors (MPAs)” reviews some of the important concepts behind the MPAs of Solstice EM. This chapter uses the sample MPA implementation included with Solstice EM to illustrate the MPA interfaces and environments.

Chapter 12 “Controlling Access to Applications and Data” explains how to make applications and data secure by controlling access to applications, application features, managed objects, event notifications, and management protocol adapters (MPAs).

Chapter 13 “Optimizing Performance” explains how to tune an application to obtain optimum performance.

Chapter 14 “Guidelines for Compiling and Linking Applications” states the compiler version requirements for applications you develop by using the Solstice EM C++ APIs. It lists, for each class in the Solstice EM C++ APIs, the header files you need to include in your application code and the libraries you need to link your applications with. This chapter also explains the flags you must set when you compile applications developed by using the Solstice EM C++ APIs.

Chapter 15 “Troubleshooting” provides guidelines on how to troubleshoot errors specific to applications developed by using the Solstice EM C++ APIs. This chapter also explains how to obtain debug information from the Solstice EM platform.

Chapter 16 “Integrating Applications With Solstice EM” explains how to add new applications to the Network Tools and Administration windows of Solstice EM. This chapter also explains how to make new applications accessible by extending the menus of existing Solstice EM tools.

Chapter 17 “Writing RPC Agents for Solstice EM” provides guidance on how to write agents by using the Site/SunNet/Domain Manager (SNM) interfaces and libraries.

Appendix A “Solstice EM C++ Source Code Examples introduces the C++ source code examples supplied with Solstice EM and provides guidelines for compiling these examples.

Appendix B “Standards Reference and Further Reading lists the standards on which Solstice EM is based. In addition, this appendix provides a list of technical terms used in the Solstice EM environment with a reference for each term to the standard in which the term is defined. This appendix also provides a list of other books that explain concepts on which Solstice EM is based.

Typographic Conventions

The following table describes the typographic conventions used in this book.

TABLE P-1 Typographic Conventions

Typeface or Symbol	Meaning	Example
AaBbCc123	The names of commands, files, and directories; on-screen computer output	Edit your <code>.login</code> file. Use <code>ls -a</code> to list all files. prompt% You have mail.
AaBbCc123	What you type, contrasted with on-screen computer output	<div>machine_name% su Password:</div>
AaBbCc123	Command-line placeholder: replace with a real name or value	To delete a file, type <code>rm filename</code> .
AaBbCc123	Book titles, new words or terms, or words to be emphasized	Read Chapter 6 in <i>User's Guide</i> . These are called <i>class</i> options. You <i>must</i> be root to do this.

Shell Prompts

The following table shows the default system prompt and superuser prompt for the C shell, Bourne shell, and Korn shell.

TABLE P-2 Shell Prompts

Shell	Prompt
C shell prompt	prompt%
C shell superuser prompt	prompt#
Bourne shell and Korn shell prompt	\$
Bourne shell and Korn shell superuser prompt	#

Accessing Sun Documentation Online

The `docs.sun.comsm` web site enables you to access Sun technical documentation on the Web. You can browse the `docs.sun.com` archive or search for a specific book title or subject at `http://docs.sun.com`.

Also, you can view the online documentation by pointing your browser to the following URL, `file:/opt/SUNWconn/em/docs/SEMDOCHP/index.html`

Sun Welcomes Your Comments

Sun is interested in improving its documentation and welcomes your comments and suggestions. You can send your comments by email to `docfeedback@sun.com`.

Please include the part number of your document in the subject line of your email.

Introduction to the Solstice EM C++ Development Environment

The Solstice Enterprise Manager (Solstice EM) C++ development environment enables you to extend the functionality of Solstice EM by developing custom applications to meet your particular network management needs.

This chapter introduces the Solstice EM C++ development environment.

- Section 1.1 “What You Can Develop in the Solstice EM C++ Development Environment” on page 1-1
- Section 1.2 “Solstice EM Network Management Model” on page 1-2
- Section 1.3 “Solstice EM Programming Model” on page 1-2
- Section 1.4 “Overview of the Application Development Process” on page 1-4

1.1 What You Can Develop in the Solstice EM C++ Development Environment

Solstice EM is a network and element management platform that simplifies the management of large and complex networks. Solstice EM is suitable for many management tasks, for example:

- Element management
- Subnetwork management
- Network management

Use the Solstice EM development environment to develop client applications to perform specific network management tasks when there is a need to:

- Present information in a specialized fashion that cannot be achieved by using existing Solstice EM components
- Manipulate information in a manner that is not possible by using Solstice EM subcomponents

1.2 Solstice EM Network Management Model

Network management in the Solstice EM environment follows the International Organization for Standardization (ISO) network management model. This model is based around manager and agent applications that exchange network management information. The ISO network management model is object oriented. According to this model, a network resource that you want to manage is represented as a managed object. A managed object is a software abstraction of a managed resource. For more information, refer to Section 2.1 “ISO Management Model” on page 2-1.

1.3 Solstice EM Programming Model

To develop robust applications that are easy to maintain, adopt a component-based programming model. Applications developed according to such a model are built from a number of separate components. Building applications from separate components simplifies the development and maintenance of applications, particularly if the application is large and complicated, or if the network to be managed is subject to change. Application development is simplified because changes to one component do not require the entire application to be modified.

Building an application from a number of different components enables you to isolate each component to deal with its own data. Isolating each component minimizes the effects on your application of changes to the network that your application will manage. If your application has to manipulate new data types (for example, because a new device type has been added to the network) you need to modify only the components that handle the new data types. The rest of your application is unaffected.

1.3.1 Solstice EM Application Program Interface (API) Component

The Solstice EM API component handles interactions between your application and the Solstice EM platform. Changes to your network can affect the Solstice EM API component. To minimize the effects of such changes, keep the Solstice EM API component independent of your network management data whenever possible. In particular, try to enable your code for this component to handle data in any format.

To enable you to write code that can handle data in any format, the Solstice EM development environment requires you to take account only of the operations permitted on managed objects. You do not need to take account of the attributes of managed objects, nor the data types of these attributes. The Solstice EM development environment enables you to query managed object definitions to obtain information about attributes of managed objects. For more information, refer to Section 5.10 “Retrieving Data From the Metadata Repository” on page 5-27.

Another means of keeping this component independent of your network management data is to decode your data and pass the decoded data to an application-specific class that stores important attributes. Chapter 9 explains how to decode complex data types.

1.3.2 Data Component

The data component provides a programmatic representation of your network management data. The data component is only necessary when the amount of network management data is large. If your application has to maintain only a subset of the data, you can enhance the performance of your application by writing C++ classes to represent the data. By keeping this representation in separate C++ classes, you minimize the effect of changes to your network management data on your application.

1.3.3 Graphical User Interface (GUI) Component

The GUI component handles interaction between your application and its users. The GUI component contains:

- Code specific to the window manager for creating widgets
- References to GUI callbacks

Isolate the code for this component from application specific code to minimize the effects on your application of changes to the GUI.

Computer-aided software engineering (CASE) tools such as SPARCworks™ Visual™ GUI builder simplify the generation of code for this component. If you use such a tool to generate code for the GUI component, isolate the generated code from code you write.

Isolating generated code simplifies maintenance and future development of your application. For example, if the look and feel of the GUI change, all you need to do is make changes in the CASE tool, generate new code and implement new functionality in the GUI component.

1.4 Overview of the Application Development Process

The application development process in the Solstice EM development environment is similar to the process that is followed on any serious software engineering project. Such a process typically consists of the following phases:

- Requirements analysis and high-level design
- Low-level design
- Implementation
- Unit testing and debugging
- Integration
- System testing

1.4.1 Requirements Analysis and High-Level Design

The requirements analysis phase identifies the functional requirements of your application. The requirements analysis phase consists of:

- Writing a problem statement
- Identifying the problem domain
- Identifying candidate managed objects
- Writing an initial version of the data dictionary
- Writing a top level description of the required functionality (for example, by using use cases)
- Defining the functionality of the application

The high-level design phase adds to the requirements specification by identifying nonfunctional requirements. The high-level design phase consists of:

- Defining interaction with other systems
- Identifying responsibilities for candidate managed objects
- Identifying relationships between candidate managed objects
- Writing the object model, for example, by using the unified modeling language (UML), or directly by using the guidelines for the definition of managed objects (GDMO)
- Updating the data dictionary

Information on how to write an object model by using GDMO is given in Chapter 2.

Special considerations in the requirements analysis and high-level design phases for an application developed by using Solstice EM include:

- Device type properties
- Network properties
- User interaction
- Management information sharing
- Access control for Solstice EM applications
- Multiple management information server (MIS) management

1.4.1.1 Device Type Properties

If you are writing an application to manage a new class of device that has been added to your network, consider which aspects of the device need to be managed. Considering the aspects of the device that need to be managed involves identifying the parameters of the device that you want to control and monitor. The parameters you identify affect the object model you write for the device.

After you have identified the aspects of the device that you want to manage, consider how your application will manage the device. Considerations for how your application will manage a device include:

- Polling a device
- Handling unsolicited messages from a device
- Setting configuration parameters of a device
- Monitoring the performance of a device
- Handling error conditions for a device

Polling a Device

If a device needs to be polled, decide whether it can use existing poll rates or new poll rates. If a polling operation detects an error, consider what action should be taken. When considering the action to take in response to an error, decide whether:

- Actions other than the default actions are required.
- The polling rate needs to be changed when an error is detected.
- An alarm needs to be raised for each error detected.

Handling Unsolicited Messages From a Device

If a device sends notifications, traps, or other unsolicited messages, consider how your application will handle such messages.

If the messages are of a type that the Solstice EM platform already recognizes, decide if the messages need to be handled differently from how the Solstice EM platform is currently configured to handle messages of that type.

If the messages are of a type unrecognized by the Solstice EM platform, decide if the messages need to be handled or can be ignored. If the messages need to be handled, decide whether they can be handled in a similar way to messages of a type that the Solstice EM platform currently recognizes, or whether they need special handling.

Setting Configuration Parameters of a Device

When you add a new class of device to a network decide how the configuration parameters of the device will be set. If a configuration parameter is set to a default value for all instances of a device, consider specifying that value in the object model of the device. If a configuration parameter is set to a different value for each instance of a device, consider enabling users of your application to set the parameter.

Monitoring the Performance of a Device

To monitor the performance of a device you need to know what is the normal performance of the device. To enable a device to report performance problems, identify the attributes or behaviors that you can use to indicate normal and abnormal performance. These attributes or behaviors are defined in the managed object class that represents the class of the device. Also determine if existing performance defaults can be used.

Handling Error Conditions for a Device

To handle error conditions for a device, identify the normal and error states of the device. After you have identified these states, assign a severity to each state. Assigning severities is a policy decision.

To enable a device to report an error condition, identify the attributes or behaviors that you can use to indicate normal and error states. These attributes or behaviors are defined in the managed object class that represents the class of the device.

To facilitate error recovery, decide to whom error conditions should be reported and how they should be reported. For example, decide whether an indication of the alarm in the Viewer tool by changing the color of an icon is sufficient. Also determine whether any standardized process or commands exist to correct error conditions associated with a device. If such processes do not exist already, consider if you need to implement them in your application.

1.4.1.2 Network Properties

The properties of the network your application will manage affect the design of your application. Considerations arising from network properties include:

- Importance of the device to the functioning of your network
- Performance requirements for your network backbone

Importance of the Device to the Functioning of Your Network

If you are managing a device, the importance of the device to your network affects how you choose to manage the device. A device that is part of your network backbone, or is a dedicated file server or application server, is likely to play a key role in your network.

If a device plays a key role in your network, consider whether you need to monitor it more closely than less important devices. For example, if the device is polled, consider whether the polling rate should be higher than for other less important devices.

If the device is a server, consider how frequently it is accessed. Consider also the effect on your network of a server failure.

Performance Requirements of Your Network Backbone

The smooth operation of a network requires that the network backbone performs adequately. When you design a network management application, determine what level of performance is required for the backbone of the network, what level of performance should be considered marginal, and what level of performance should generate alarms.

1.4.1.3 User Interaction

The needs of users who will interact with your application affect the design of your application. Taking account of the needs of users involves:

- Deciding how information is presented to users of your application
- Identifying information that is presented to users of your application
- Determining how users should start your application
- Preventing users from introducing errors

Identifying Information That is Presented to Users of Your Application

To manage a network, users of your application need to be informed of the state of network resources. Analyze the functional requirements of your application to identify information that needs to be presented to users of your application.

When you have identified information that needs to be presented, analyze the network that your application will manage to find out where the information will come from. Find out, for example, if the information will be provided by a device or by another application, such as the Nerve Center or event forwarding discriminators (EFDs).

Determine if you need to gather, summarize, or process information in a manner that is not possible by using Solstice EM subcomponents such as Nerve Center requests, EFDs, or log objects. Where possible, use Solstice EM subcomponents to obtain the information you want to present to users of your application. Using Solstice EM subcomponents saves the costs of having to develop entire applications from scratch.

Deciding How Information is Presented to Users of Your Application

To determine how best to present information to users of your application, analyze how that information will be used. Determine whether the information needs to be presented in a specialized manner that is not possible by using existing Solstice EM tools, such as the Viewer or the Alarm Manager. Specialized presentation of information includes special presentation windows, GUI-based device front ends, or terminal output.

Even if you require specialized presentation of information, consider if it also makes sense to display some information in standard Solstice EM tools such as the Viewer, the Log Manager, or the Alarm Manager, or in another Solstice EM client application you have developed.

Determining How Users Should Start Applications You Develop

Applications you develop will typically be used in conjunction with other Solstice EM components to provide a complete network management solution. To enhance the usability of your network management solution, determine which is the most convenient means for users to start your applications. Depending on the purpose of an application, you can enable users to start the application from:

- The Network Tools window
- The Administration window
- A menu in another Solstice EM tool
- A Nerve Center request.

Identify the information that your application needs when it is started, for example context information or initialization information. When you have identified this information, determine where it comes from and whether it differs from the information your application normally uses or displays.

Preventing Users From Introducing Errors

Design the user interface of your application to prevent users from introducing errors. Where possible, prevent users from carrying out sequences of operations that will introduce error conditions into your network.

Control access to critical data to ensure that such data is modified only by operators who are qualified to do so. For more information, refer to Section 1.4.1.5 “Access Control for Solstice EM Applications” on page 1-10.

1.4.1.4 Management Information Sharing

If your application needs to share management information, considerations for your application include:

- Access to information from multiple users
- Information sharing and storage requirements
- Distribution of information in unsolicited messages
- Access control for shared information

Access to Information From Multiple Users

How multiple users access information affects the design of your application. If your application is intended to be used by more than one user at a time, you need to enable your application to support multiple concurrent users.

You also need to determine if several applications need to access to management information simultaneously. If the information that needs to be shared exists in the Solstice EM MIS, all you need to do is develop an application to access the information.

Information Sharing and Storage Requirements

If your application will gather or summarize information, determine whether this information needs to be stored permanently or temporarily. If the information needs to be stored temporarily, determine if it needs to be stored only as long as your application is running, or as long as a server that serves your application is running.

Identify the sources of the information your application will gather or summarize. For example, the information may reside in the MIS, EFDs, or the Nerve Center.

If other applications require information gathered or summarized by your application, determine how best to share this information.

Note – Applications normally share information by using the MIS. To enable applications to share information directly (that is, without using the MIS), use the application-to-application API. For an overview of the Solstice EM C++ APIs, see Section 1.4.2.1 “API Choice” on page 1-14.

Distribution of Information in Unsolicited Messages

If several copies of your application will be running simultaneously, determine if all copies need to receive notifications, traps, or other unsolicited information.

Access Control for Shared Information

If information is shared between several copies of your application, determine if the information used by one copy of the application needs to be kept secure from other copies of the same application. Similarly, if information is shared between applications, determine if the information used by an application needs to be kept secure from other applications.

Note – Solstice EM does not currently support this type of data partitioning for security purposes.

1.4.1.5 Access Control for Solstice EM Applications

If you want to enforce access control for your application, you need to decide which of the following levels of access control you require:

- Application-level access control
- Application-feature-level access control
- Managed-object-level access control
- Event notification access control
- Management protocol adapter (MPA) access control

For more information about designing access control for your applications, refer to Chapter 12.

Application-Level Access Control

Implement application-level access control if you want your entire application to be inaccessible to some users of the network management solution that your application is a part of. For example, if your application is used for the administration of your network management solution, make the application accessible only to system administrators and inaccessible to network operators.

If you implement application-level access control, make sure that your application gives proper feedback to a user that is denied access to your application.

Application-Feature-Level Access Control

Implement application-feature-level access control if you want some users to be able to access some, but not all, the features of your application. For example, if your application enables users to monitor, add, modify, and delete network resources, implement application-feature-level access control to allow some users to monitor network resources, but not to add, modify, or delete network resources.

If you implement application-feature-level access control, make sure that your application gives proper feedback if a user is denied access to a feature. Where possible, make sure that your application prevents users from performing operations they do not have permission to perform. In a graphical application, make commands for performing such operations inactive and grayed out.

If you implement application-feature-level access control, make the list of application features available to your system administrator so that the system administrator can grant users access rights to perform various operations.

Managed-Object-Level Access Control

Implement managed-object-level access control if you want some managed objects to be inaccessible to some users of your network management solution.

Managed-object-level access control denies users access to managed objects regardless of which application they use to try to access the managed objects. If you use application-feature-level access control to deny access to these managed objects, you do not prevent users from accessing the managed objects by using other features of other applications.

If you implement managed-object-level access control, make sure that your application gives proper feedback if a user is denied access to a managed object. In addition, make sure that your application can handle any exceptions or errors thrown if a user is denied access to a managed object.

Event Notification Access Control

Implement event notification access control if you want to ensure that a user's event logs contain only event notifications emitted by managed objects to which the user has access. By default, all events that the Solstice EM platform receives are written to a user's event logs, including event notifications from managed objects that the user is normally denied access to.

MPA Access Control

Implement MPA access control if you want some managed objects that are accessed through an MPA to be inaccessible to some users of your application.

If you implement MPA access control, make sure that your application gives proper feedback if a user is denied access to a managed object accessed through an MPA.

1.4.1.6 Multiple Management Information Server (MIS) Management

If more than one Solstice EM MIS will be used to hold your network management information, you need to consider how multiple MISs will be managed. Managing multiple MISs involves:

- Determining the subordinate objects of the root of the MIT
- Deciding if the fully distinguished name (FDN) table requires manual updating
- Assigning managed objects to an MIS
- Deciding which information is exchanged between MISs

Determining the Subordinate Objects of the Root of the MIT

In the Solstice EM environment, managed objects that represent your network resources are arranged in a hierarchy known as the management information tree (MIT). For more information on the MIT, refer to Section 2.2.6 “Identifying Containment Relationships” on page 2-14.

If more than one EM MIS will be used to hold your network management information, determine which managed objects will be located under the root of the MIT. For each managed object that will be located under the root of the MIT, you need to determine its fully distinguished name (FDN). For more information on FDNs, refer to Section 2.2.6.2 “Names of Managed Object Instances” on page 2-18.

Deciding if the FDN Table Requires Manual Updating

Each managed object that resides outside the local MIS has an entry in the FDN table. The FDN table maps the FDN of a remote managed object to the location of the entity in which the managed object resides. The location of the remote entity is given by its presentation selector.

If more than one EM MIS will be used to hold your network management information, you need to determine if the FDNs of remote managed objects need to be manually updated into the FDN table.

Assigning Managed Objects to an MIS

If more than one EM MIS will be used to hold your network management information, you need to assign each managed object to an MIS.

Deciding Which Information is Exchanged Between MISs

If more than one EM MIS will be used to hold your network management information, you need to decide on the types of information that will be automatically passed from one MIS to another.

When information is shared between MISs, you need to decide how MISs will be arranged hierarchically. The possible arrangements are as follows:

- One MIS acts as a manager. In this arrangement, all other MISs pass information to this manager, or within the MIT one MIS contain other MISs.
- All MISs act as peers. In this arrangement, certain types of information are shared with most or all other MISs.
- One MIS acts as a manager and some or all of the remaining MISs act as peers.

1.4.2 Low-Level Design

The low-level design phase follows the high-level design phase. The low-level design phase defines how your application will meet the requirements identified in the requirements analysis and high-level design phases. The low-level design phase consists of:

- Introducing supporting elements that will enable your application to function
- Adding design details to the object model
- Performing systems design
- Specifying interfaces
- Specifying classes
- Defining an implementation strategy for your system

Special considerations in the low-level design phase for an application developed by using Solstice EM include:

- API choice
- Object location
- Provision of behavior code for objects
- Identification of managed objects

1.4.2.1 API Choice

The Solstice EM C++ development environment provides a number of APIs, each of which has a specific purpose. You are free to use any combination of the APIs in a single application. The choice of APIs to use in an application depends on a combination of factors, such as:

- The functional requirements of your application
- The importance of keeping coding simple
- The performance requirements of your application

The architecture of the C++ APIs supplied with Solstice EM is shown in FIGURE 1-1.

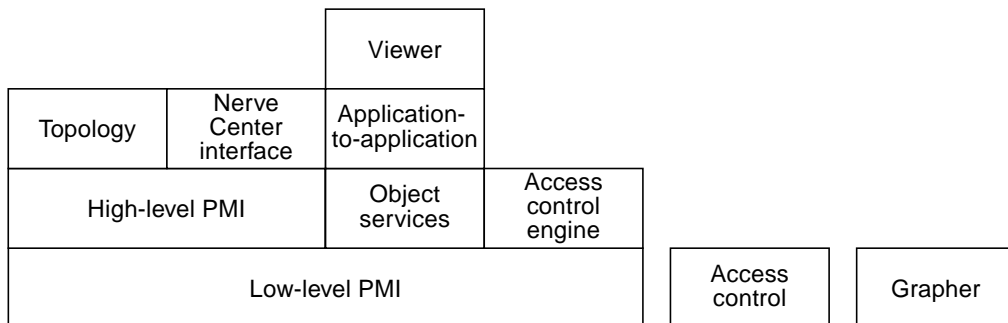


FIGURE 1-1 Architecture of the Solstice EM C++ APIs

Low-Level Portable Management Interface (PMI)

The low-level Portable Management Interface (PMI) provides a low-level abstraction of management services. The management services provided by the low-level PMI are equivalent to the services defined in ITU-T X.710/ISO-9595 *Common Management Information Services (CMISE)*. The interface provided by the low-level PMI is distributed and independent of any transport-layer protocols.

Use the low-level PMI for applications that require high performance. Using the low-level PMI requires you to write more code than using the high-level PMI.

High-Level PMI

The high-level PMI is the primary API for management applications. This API provides a high-level abstraction of the managed resources in a network. This abstraction provides a means for manipulating objects that is independent of the class description, supported protocol, or location of the managed objects. The generic nature of the high-level PMI eases the development of network management applications.

Use the high-level PMI to keep coding simple for applications that do not require optimum performance.

The high-level PMI is built on the low-level PMI.

Object Services API

The object services API enables object behavior functions you develop to access information and services provided by the MIS. The decision to use these services depends on the behavior defined for an object. To develop object behavior functions, use the Solstice EM object development tools (ODT).

The object services API is built on the low-level PMI.

Topology API

The topology API provides access for management applications to topology services provided by Solstice EM. This API enables you to manipulate topology nodes, which are displayed in the Network Views tool. For information on the Network Views tool, refer to *Managing Your Network*.

The topology API hides the topology implementation, enabling you to port topology-based applications easily to future releases of Solstice EM as the high-level PMI evolves.

The topology API is built on the high-level PMI.

Nerve Center Interface

The Nerve Center interface enables provides an API to the Nerve Center. The Nerve Center interface enables you to create request templates, launch requests against objects in the MIS, and retrieve information about objects. For information on the Nerve Center, refer to the *Customizing Guide*.

The Nerve Center interface is built on the high-level PMI.

Application-to-Application API

The application-to-application API enables applications to share information directly (that is, without using the MIS). Applications normally share information by using the MIS. The application-to-application API is built on the high-level PMI.

Viewer API

The viewer API enables you to integrate applications with the Network Views tool. This API enables applications to communicate with and modify the Network Views tool. For example, an application can get the current view, set the contents of the Network Views footer, or change the Network Views zooming or magnification.

The viewer API also enables applications to register with the Network Views tool to receive events generated by the tool. When an application registers with the Network Views tool to receive events, the application also registers callbacks that are executed when events are received from the Network Views tool.

For information on the Network Views tool, refer to *Managing Your Network*.

The viewer API is built on the application-to-application API.

Access Control Engine API

The access control engine API enables you to control user access to objects supported via user-developed MPAs and auxiliary servers. In this way, the access control engine API enables user-created MPAs and auxiliary servers to impose access control on the objects they manage.

For more information about designing access control for your applications, refer to Chapter 12.

The access control engine API is built on the low-level PMI.

Access Control API

The access control API enables you to control user access to your application, features within your application, and managed objects manipulated by your application. This API provides classes and member functions that enable you to assign access control rules to groups of users. These classes and member functions also enable you to define access control rules.

For more information about designing access control for your applications, refer to Chapter 12.

Grapher API

The grapher API enables your application to send data to the Grapher tool. If the Grapher tool is not running, the grapher API starts it automatically. The grapher API supports:

- **Static graphs.** A static graph cannot be updated after it is created.
- **Dynamic graphs.** A dynamic graph can be updated after it is created. Use a dynamic graph to plot a variable that changes with time.

For information on the Grapher tool, refer to *Managing Your Network*.

1.4.2.2 Object Location

In the Solstice EM environment, managed objects that represent your network resources are stored as objects. Objects are either local or remote.

- **A local object** resides in the MIS. The operations supported by a local object are carried out by the MIS processes. Attribute values of the local object are stored or maintained in the MIS or its persistent store. The MIS maintains a reference to the local object instance in the MIT.
- **A remote object** resides outside the MIS. The MIS maintains a reference to a remote object instance via the MIT. A remote object typically resides in an agent, or in another MIS.

1.4.2.3 Provision of Behavior Code for Objects

All objects need behavior code. Behavior code enables an object to respond to management requests.

The MIS provides behavior code only for local objects. The MIS does not provide behavior code for remote objects. You must ensure that the entity within which a remote object resides provides behavior code for the object.

The behavior code that the MIS provides enables an object in the MIS to exhibit default behavior. If you want a local object to exhibit custom behavior, you can develop custom behavior code by either of the following means:

- Using ODT
- Writing an MPA

Using ODT

Using ODT simplifies the development of custom behavior code by generating much of the code for you. However, an object the behavior of which is developed by using ODT *must* reside in the MIS. Use ODT with care because errors in the behavior code of such an object may cause the MIS to fail.

For information on using ODT, refer to Chapter 10.

Writing an MPA

An MPA performs protocol translation required for communication between the Solstice EM platform and an external entity, such as an agent. Writing an MPA requires you to write more code than using ODT. However, an object the behavior of which is implemented by an MPA can reside on a separate server from the MIS, thereby enhancing the performance and reliability of your management solution.

For information on how to write an MPA, refer to Chapter 11.

1.4.2.4 Managed Object Identification

By default, a managed object in the Solstice EM environment is identified by its FDN as explained in Section 2.2.6.2 “Names of Managed Object Instances” on page 2-18. In an MIT with many levels of containment, FDNs become long and complicated. The FDNs of managed objects that are many levels below the root of the MIT are particularly long and complicated. To simplify the task of selecting managed objects you can assign nicknames to managed objects and select managed objects by specifying their nicknames.

For more information on setting up nicknames, see Section 5.3.2 “Selecting a Managed Object by Specifying its Nickname” on page 5-10.

1.4.3 Implementation

The implementation phase follows the low-level design phase. During the implementation phase, code that implements the design of your application is written, and the executable files of your application are generated.

In the Solstice EM C++ development environment, the implementation phase consists of:

- Enabling applications to access managed objects
- Handling errors
- Performing operations on managed objects
- Performing management operations on object collections
- Handling events
- Performing asynchronous management operations
- Encoding and decoding complex ASN.1 values
- Controlling access to applications and data
- Optimizing performance
- Compiling and linking applications

1.4.3.1 Enabling Applications to Access Managed Objects

To manage a network, the applications you develop must have access to current data about managed resources. In the Solstice EM environment, managed resources are represented as managed objects. Your applications must be able to access managed objects to obtain the data they require.

For information on how to enable applications to access managed objects, refer to Chapter 3.

1.4.3.2 Handling Errors

Users need to know when an attempted network management operation has failed. By providing accurate information on why the operation failed, your applications can ease a user's work by indicating the corrective action required when problems occur.

For information on how to handle errors, refer to Chapter 4.

1.4.3.3 Performing Operations on Managed Objects

An application manages a network by monitoring and controlling managed resources in the network. In the Solstice EM environment, managed resources are represented as managed objects. An application monitors and controls managed resources by performing operations on managed objects.

For information on how to perform operations on managed objects, refer to Chapter 5.

1.4.3.4 Performing Management Operations on Object Collections

An object collection is a group of managed objects that your application can treat as a single entity. An object collection simplifies bulk operations by enabling you to select multiple managed objects to be the subject of a management operation. Any management operation that your application performs on an object collection is performed on every managed object in the object collection.

For information on how to perform management operations on object collections, refer to Chapter 6.

1.4.3.5 Handling Events

Any network management application that monitors and controls managed resources on a network needs to process information it receives from those managed resources. Such information is contained in event notifications. An event notification is an unsolicited message sent from a managed object that represents a managed resource. Event notifications contain error information and other types of status information.

For information on how to handle events, refer to Chapter 7.

1.4.3.6 Performing Asynchronous Management Operations

A management operation can take a significant length of time to finish, particularly if the operation exchanges a large quantity of data between your application and the network resources it is managing. If your application is blocked while waiting for an operation to finish, the application may appear unresponsive to a user, or may fail to respond quickly enough to important events on your network. Performing asynchronous management operations enables an application to continue with other processing without waiting for the operations to finish.

For information on how to perform asynchronous management operations, refer to Chapter 8.

1.4.3.7 Encoding and Decoding Complex ASN.1 Values

In the Solstice EM environment, attribute values in management requests, responses and event notifications are represented in a machine-independent format for transmission over a network. The format used is defined in ITU-T X.208/ISO-8824 *Specification of Abstract Syntax Notation One (ASN.1)*. This standard defines several complex data types and enables you to define your own custom data types. When an application sends a request to set an attribute value represented by a complex data type, the application must encode this value for transmission over a network. When an application receives an attribute value represented by a complex data type (for example in a response or an event notification) the application must decode this value to extract the information the value contains.

For information on how to encode and decode complex ASN.1 values, refer to Chapter 9.

1.4.3.8 Controlling Access to Applications and Data

Controlling access to applications and data prohibits unwanted access to critical applications and network components. Without access control, any user of your network management solution can read or modify all your network management and configuration data. The risks of this approach can be devastating when users without the proper authority or expertise modify your network management data or the configuration data of your network management solution. By controlling user access, users are allowed to access only those applications and data they need based on their network management responsibilities and other relevant criteria.

For information on how to control access to applications and data, refer to Chapter 12.

1.4.3.9 Optimizing Performance

The high-level PMI provides many features that simplify the coding of an application. However, if you need fast response from an application, or if an application is controlling and monitoring a large number of managed objects, you need to tune the application to obtain optimum performance.

For information on how to optimize the performance of your applications, refer to Chapter 13.

1.4.3.10 Compiling and Linking Applications

Applications you develop by using the Solstice EM C++ APIs require specific flags to be set at compilation time. You also need to link your applications with the Solstice EM C++ libraries.

For guidelines on compiling and linking applications developed by using the Solstice EM C++ APIs, refer to Chapter 14.

1.4.4 Unit Testing and Debugging

The unit testing and debugging phase follows the implementation phase. The unit testing and debugging phase assures the quality of the application by ensuring that the application meets its stated requirements.

The Solstice EM C++ development environment provides tools to help you test and debug your applications. For information on how use these tools, refer to Chapter 15. This chapter also provides guidelines on how to avoid and correct errors specific to applications developed by using the Solstice EM C++ APIs.

1.4.5 Integration

The integration phase follows the unit testing and debugging phase. The integration phase integrates your custom applications with the Solstice EM platform to create a complete network management solution.

For information on how to integrate applications with the Solstice EM platform, refer to Chapter 16.

1.4.6 System Testing

The system testing phase follows the integration phase. The system testing phase assures the quality of your complete network management solution.

Modeling Managed Objects

Network management in the Solstice EM environment follows the ISO network management model. This model is object oriented. To manage your resources in the Solstice EM environment, you need to write an object model of those resources. The object model defines the characteristics of resources your application will manage. Having written your object model you need to make it available to Solstice EM.

This chapter explains how to write an object model and make it available to Solstice EM.

- Section 2.1 “ISO Management Model” on page 2-1
- Section 2.2 “Designing the Object Model” on page 2-5
- Section 2.3 “Abstract Syntax Notation #1 (ASN.1)” on page 2-23
- Section 2.4 “Assigning Unique Identifiers” on page 2-30
- Section 2.5 “Obtaining GDMO and ASN.1 Specifications for Objects” on page 2-35
- Section 2.6 “Making Your Object Model Available to Solstice EM” on page 2-36

2.1 ISO Management Model

Network management in the Solstice EM environment follows the ISO network management model. This model is based around manager and agent applications that exchange network management information. The ISO network management model is illustrated in FIGURE 2-1.

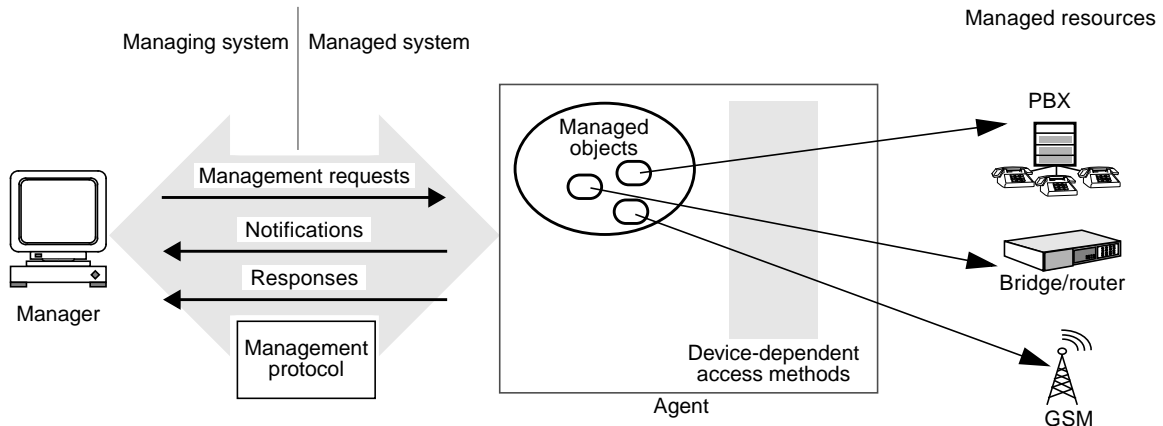


FIGURE 2-1 ISO Network Management Model

The main parts of the ISO network management model are introduced in the following subsections.

2.1.1 Managers

A manager issues management requests to one or more agents.

A manager receives information from agents in the form of:

- **Notifications.** A notification is an unsolicited message sent to manager to indicate that a change has occurred to a managed resource.
- **Responses.** A response is a message sent in response to a management request. A response contains the result of the management request, for example a confirmation that the request was carried out, or the information the manager requested.

A manager typically performs the following additional functions:

- Collecting and filtering information from agents
- Presenting information to operators of a managing system

A manager resides in a managing system.

2.1.2 Agents

An agent acts as an intermediary between a manager and managed resources. An agent receives requests from a manager. An agent sends responses to requests and issues notifications. Each agent in a managed system is responsible for carrying out management directives to control or return information from managed resources.

An agent can reside in a managed resource, or be located elsewhere and operate remotely.

2.1.3 Managed Resources

A managed resource is any network resource that can be managed. The resource can be a physical device such as a host, server, router, or subnet, or it can be a conceptual entity such as a line, a queue, or some other aspect of network operation that needs to be managed.

2.1.4 Managed Objects

The ISO management model on which Solstice EM is based is object oriented. According to this model, a managed resource is represented as a managed object. A managed object is a software abstraction of a managed resource. The managed object presents information needed to manage the resource. A managed resource may be represented by a single managed object, or by several managed objects. An agent typically contains or provides views of many managed objects.

2.1.5 Management Protocols

A management protocol is a set of rules that specify how information shall be exchanged between two entities that are communicating, such as a manager and an agent. A management protocol provides the common language required to enable managers and agents to exchange information.

A management protocol defines:

- Types of messages that agents and managers are allowed to issue
- The syntax and encoding of each type of message

Solstice EM supports the following management protocols:

- Common Management Information Protocol (CMIP)
- Simple Network Management Protocol (SNMP)
- SunNet Manager™ remote procedure call (RPC)

If the agents you want to manage use management protocols that Solstice EM supports, you do not need to take account of the management protocol when you use Solstice EM to develop applications.

Management protocol adapters (MPAs) perform protocol translation required to enable Solstice EM to support several management protocols. If you want to support a management protocol for which Solstice EM does not provide an MPA, you can extend Solstice EM by writing your own MPA for that protocol. For information on how to write an MPA, refer to Chapter 11.

2.1.6 Manager-Agent Hierarchy

Managers and agents are designed to be deployed in a hierarchy. A manager can also act as an agent to a higher-level system. This allows control of the network to be distributed throughout the network, while maintaining overall control of those aspects that are best managed centrally.

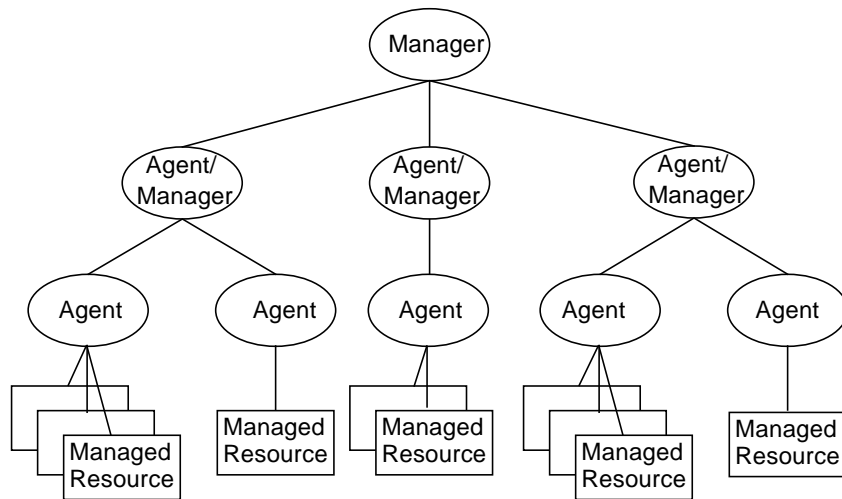


FIGURE 2-2 Manager-Agent Hierarchy

2.2 Designing the Object Model

An object model defines the managed objects you want to control and monitor. It identifies:

- The features of managed objects that you want your application to control and monitor
- The control and monitoring operations you want to perform
- The relationships between the managed objects that you want your application to control and monitor

The object model is shared by the manager and any agents that the manager manages. If you are writing a manager for an existing agent, use the object model developed for the agent. In this case, do not write an object model specially for the manager.

To define managed objects in the Solstice EM environment, use the notation defined in ITU-T X.722/ISO-10165-4 *Guidelines for the Definition of Managed Objects (GDMO)*.

This recommendation defines templates for the definition of managed objects. These templates specify the elements that should be included in the definition and the notation used to express each element. For information on the format of each template, refer to Appendix C.

To use GDMO notation to define managed objects, fill out the templates with relevant information and by using the appropriate syntax. Filling out GDMO templates involves:

- Defining the task
- Identifying managed object classes
- Identifying inheritance relationships
- Identifying the characteristics of a managed object class
- Describing the behavior of items in the object model
- Identifying containment relationships
- Grouping information into packages
- Identifying a GDMO definition

Solstice EM accepts GDMO specifications written by using a GDMO modeling tool such as Solstice GDMO Builder. EM also accepts GDMO specifications generated from Unified Modeling Language (UML) modeling tools.

2.2.1 Defining the Task

Defining the task identifies the management operations your application will perform. Defining the task involves:

- Identifying the monitoring and control operations you want your manager to perform on the resource
- Verifying the capabilities of the resource you want to manage

2.2.1.1 Identifying Monitoring and Control Operations - Example

The satellite example illustrates applications to manage satellites that broadcast several television channels, with each channel being received by one or more satellite dishes.

For a satellite, the following need to be monitored:

- Which channels the satellite is broadcasting
- The amount of data the satellite has to handle
- The quality of the signal to the satellite
- Whether the satellite is disabled or enabled

For a satellite, the following need to be controlled:

- The position of the satellite
- Whether the automatic navigation system for the satellite is engaged
- What happens to the satellite if the company is subject to hostile takeover effort
- Whether the satellite is locked, unlocked, or shutting down

For a channel, the following need to be monitored:

- Which dishes are receiving the channel
- The amount of data the channel has to handle
- The quality of the signal broadcast on the channel
- Whether the channel is disabled or enabled

For a channel, the following need to be controlled:

- Whether the main or backup transmitter should be used to broadcast the channel
- Which program is being broadcast on the channel
- How long an operator has to censor inappropriate scenes in a program broadcast on the channel
- Whether the channel is locked, unlocked, or shutting down

For a dish, the system needs to control whether access to the current program is blocked for the dish.

2.2.1.2 Verifying the Capabilities of the Managed Resource

When you have identified what you want to do, find out if it is possible. For example, check if you can find out the amount of data a satellite has to handle. To check, refer to developer's documentation, standards and other reference material for the resource you want to manage. If some of the information you require is not available, either modify your requirements accordingly, or put in place a way of obtaining the information.

2.2.2 Identifying Managed Object Classes

Managed objects are defined in terms of managed object classes. A managed object class is a definition of how all managed objects of a particular type should be implemented. Individual managed objects are referred to as instances of a class. An object class is defined once and reused thereafter for all objects of the same class.

Identify the managed object classes you need to represent your managed resources. In general, there should be one managed object class for each physical device you want to manage. You may also require a managed object class for each conceptual entity that you want to manage.

The purpose of the applications in the satellite example is to manage satellites that broadcast several television channels, with each channel being received by one or more satellite dishes. From this analysis, the following physical devices and conceptual entities can be identified:

- Satellite physical device
- Channel conceptual entity
- Dish physical device

CODE EXAMPLE 2-1 shows the definition of the `dish` managed object class in the satellite sample programs, expressed in GDMO notation.

CODE EXAMPLE 2-1 GDMO Definition of the `dish` Managed Object Class

```
...
dish MANAGED OBJECT CLASS
    DERIVED FROM "Rec. X.721 | ISO/IEC 10165-2 : 1992" : top;
    CHARACTERIZED BY
        dishPackage;
...
```

This example shows that the `dish` managed object class:

- Is derived from, or inherits the characteristics of, a class named `top` defined in ITU-T X.721/ISO-10165-2 *Definition of Management Information*

- Is characterized, or defined, by the packages it contains, in this case the `dishPackage` package

2.2.3 Identifying Inheritance Relationships

New classes can be defined in terms of existing classes. The new class is a subclass of the class from which it is derived and may in turn have subclasses of its own. The class from which it is derived is called its superclass.

A subclass inherits all of the characteristics of its superclass. The inherited characteristics of the subclass can then be extended by adding new attributes, actions and notifications. In the ISO management model, it is not possible to delete any of the characteristics of the superclass.

The ultimate superclass is the `top` object class, from which all other object classes are derived. This object class is specified by the OSI systems management model in ITU-T X.721/ISO-10165-2 *Definition of Management Information* and contains definitions for the attributes that are common to all object classes.

Whenever possible, derive the managed object classes you require from standard managed object classes, or from managed object classes that have already been generated for other purposes. In the satellite example, all three managed object classes inherit characteristics from the `top` managed object class. The inheritance tree for the satellite example is shown in FIGURE 2-3.

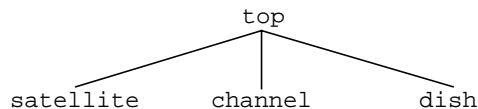


FIGURE 2-3 Example Inheritance Tree

2.2.4 Identifying the Characteristics of a Managed Object Class

After you have identified the managed object classes you need, identify the characteristics of each managed object class. To identify the characteristics of a managed object class, think about how the information you want to retrieve, and the operations you want to perform, relate to the managed object class.

Identifying the characteristics of a managed object class involves creating, attributes, notifications, and actions.

2.2.4.1 Attributes

An attribute is a data element that is encapsulated in a managed object. Attributes reflect the state information that applies to a managed object. Each attribute corresponds to one of the characteristics of the resource that the managed object represents. An attribute has a name, a type, and one or more values that reflect the current status of the associated resource.

Note – The management operations permitted for an attribute are defined in the package definition of the managed object class that contains the attribute, not in the definition of the attribute itself. For more information, see Section 2.2.7 “Grouping Information Into Packages” on page 2-20.

In the satellite example, attributes are required for the `satellite`, `channel` and `dish` managed object classes as follows:

- `satellite` - TABLE 2-1
- `channel` - TABLE 2-2
- `dish` - TABLE 2-3

TABLE 2-1 Attributes for the `satellite` Managed Object Class

Name	Purpose
<code>satelliteId</code>	Represent the name of a satellite
<code>activeChannels</code>	Monitor the channels that a satellite is broadcasting
<code>packetsReceived</code>	Monitor the total number of packets that a satellite has received
<code>packetRetries</code>	Monitor the quality of the signal to a satellite
<code>packetsSent</code>	Monitor the number of packets sent to a satellite
<code>operationalState</code>	Monitor the operational state (enabled or disabled) of a satellite
<code>altitude</code>	Control the distance of a satellite from the earth
<code>coordinates</code>	Control the position of a satellite
<code>selfDestructCode</code>	Represent a code for instructing a satellite to be destroyed
<code>spaceJunkAvoidance</code>	Control whether the automatic navigation system for a satellite is engaged
<code>administrativeState</code>	Control whether a satellite is locked, unlocked, or shutting down

TABLE 2-2 Attributes for the `channel` Managed Object Class

Name	Purpose
<code>channelId</code>	Represent the name of a channel
<code>activeDishes</code>	Monitor the dishes that are receiving a channel
<code>packetsReceived</code>	Monitor the total number of packets a channel has received
<code>packetRetries</code>	Monitor the quality of the signal to a channel
<code>packetsSent</code>	Monitor the number of packets sent to a channel
<code>operationalState</code>	Monitor the operational state (enabled or disabled) of a channel
<code>coordinates</code>	Control the position of the satellite broadcasting a channel
<code>backupCoordinates</code>	Control whether the main or backup transmitter should be used to broadcast a channel
<code>program</code>	Control the program that is being broadcast on a channel
<code>transmitDelay</code>	Control how long an operator has to censor inappropriate scenes in a program broadcast on a channel
<code>administrativeState</code>	Control whether a channel is locked, unlocked, or shutting down

TABLE 2-3 Attributes for the `dish` Managed Object Class

Name	Purpose
<code>dishId</code>	Represent the name of the dish
<code>vchipId</code>	Represent an identifier for controlling access to programs that the dish receives
<code>censureButton</code>	Control whether access to the current program is blocked for the dish
<code>coordinates</code>	Control the position of the satellite broadcasting to the dish

CODE EXAMPLE 2-2 shows the definition of the `censureButton` attribute.

CODE EXAMPLE 2-2 GDMO Definition of the `censureButton` Attribute

```
...
censureButton ATTRIBUTE
    WITH ATTRIBUTE SYNTAX SAT-MAN-ASN1.ButtonPress;
    MATCHES FOR EQUALITY;
    BEHAVIOUR censureButtonBehaviour BEHAVIOUR DEFINED AS
        ! This attribute indicates whether the customer
        watching TV has activated the censure button.!!;
    ;
    REGISTERED AS { satman-attribute 16 };
...
```

In this example, the attribute named `censureButton` indicates whether the customer has activated the censure button. The type of this attribute is a custom-defined ASN.1 type named `ButtonPress`. This ASN.1 type is defined in an ASN.1 module named `SAT-MAN-ASN1`.

The ASN.1 syntax definition of `ButtonPress` is given in CODE EXAMPLE 2-3.

CODE EXAMPLE 2-3 ASN.1 Syntax Definition of the `ButtonPress` Data Type

```
ButtonPress ::= ENUMERATED {
    off          (0),
    on           (1)
}
```

In this example, the `censureButton` attribute is an enumerated type. Its valid values are `off` (0), and `on` (1).

For information on ASN.1, refer to Section 2.3 “Abstract Syntax Notation #1 (ASN.1)” on page 2-23.

2.2.4.2 Actions

An action is an operation that cannot be modelled by a pre-defined operation such as getting or setting an attribute. An action enables you to implement specialized behavior, for example, changing attributes of one or many objects in a single operation or providing the results of a query in a particular format.

No actions are defined for any of the managed object classes in the satellite example.

2.2.4.3 Notifications

A notification is an unsolicited message sent from a managed object that represents a managed resource. A managed object generates a notification when the application managing it needs to know that something has changed.

Managed objects can issue notifications in response to internal and external events. For example, a managed object may issue a notification in response to a timer timing out.

Notifications can be transmitted to manager applications in the form of event notifications, or logged internally. The type of notifications issued by a managed object and the conditions under which notifications are issued form part of its definition.

In the satellite example, notifications are required as shown in TABLE 2-4.

TABLE 2-4 Notifications for the Satellite Example

Notification	Purpose
objectCreation	Signal when a new satellite, channel, or dish is added to the network management environment
attributeValueChange	Signal when an attribute of a satellite, channel, or dish changes
objectDeletion	Signal when a satellite, channel, or dish is removed from the network management environment

These notifications are defined in ITU-T X.721/ISO-10165-2 *Definition of Management Information*. For a complete list of notifications defined in ITU-T X.721/ISO-10165-2 *Definition of Management Information*, refer to TABLE 7-1.

CODE EXAMPLE 2-4 shows the GDMO specification of the objectCreation notification.

CODE EXAMPLE 2-4 GDMO Specification of the objectCreation Event

```
objectCreation    NOTIFICATION
  BEHAVIOUR      objectCreationBehaviour;
  WITH INFORMATION SYNTAX Notification-ASN1Module.ObjectInfo
    AND ATTRIBUTE IDS
      sourceIndicator      sourceIndicator,
      attributeList        attributeList,
      notificationIdentifier notificationIdentifier,
      correlatedNotifications correlatedNotifications,
      additionalText        additionalText,
```

CODE EXAMPLE 2-4 GDMO Specification of the objectCreation Event (*Continued*)

```
additionalInformation      additionalInformation;

REGISTERED AS      {joint-iso-ccitt ms(9) smi(3) part2(2) notification(10) 6};
-- changed by Technical Corrigendum 2

objectCreationBehaviour
BEHAVIOUR
    DEFINED AS "This notification type is used to report the creation of a
                managed object to another open system.";
```

Note – The notifications supported by a managed object are defined in the package definition of the managed object class. For more information, see Section 2.2.7 “Grouping Information Into Packages” on page 2-20.

2.2.5 Describing the Behavior of Items in the Object Model

The behavior of an item in the object model describes how the item reacts to internal and external events. It indicates the purpose of the item. The description of an item’s behavior is similar to a comment in a programming language.

CODE EXAMPLE 2-5 shows the GDMO specification of the behavior of the packetRetries attribute.

CODE EXAMPLE 2-5 Behavior of the packetRetries Attribute

```
packetRetries ATTRIBUTE
...
    BEHAVIOUR packetRetriesBehaviour BEHAVIOUR DEFINED AS
        ! Contains the total number of packet retries for
        the day. This attribute is used by the satellite
        to monitor the transmission quality to the
        channel. If the number of retries increases
        quickly, the satellite will automatically switch
        transmissions to the channel’s back-up location!;
    ;
...
```

2.2.6 Identifying Containment Relationships

To enable applications to locate managed objects, the ISO model arranges objects in a hierarchical structure. This structure is called the management information tree (MIT), or the containment tree. One object can contain another. A containing (or superior) object may, in turn, be contained in another object. A superior object can contain more than one object, but a contained (or subordinate) object can only be contained in one superior object at a time. This restriction forces a tree structure on the hierarchy.

The ISO model defines a containment relationship in which the object *below* the root object is called the `system` object.

To design the containment tree for your object model, consider how managed objects relate to one another. The containment tree for the satellite example is shown in FIGURE 2-4.

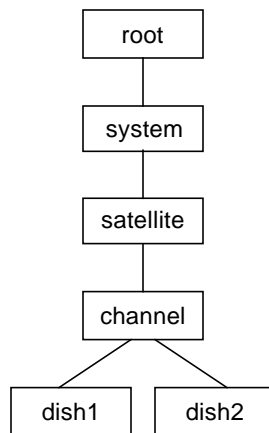


FIGURE 2-4 Example MIT

The containment tree is enforced by the naming scheme for managed object instances. The naming scheme uses name bindings to enable the relative and fully distinguished names of managed object instances to be computed.

2.2.6.1 Name Bindings

In GDMO, the naming scheme for managed object instances is defined by a name binding for a pair of managed object classes. A name binding defines the containment relationship between instances of each class in the pair. A name binding also provides additional definitions that govern the creation and deletion of managed objects.

Definition of a Containment Relationship

A name binding defines the containment relationship between instances of each class in a pair of by defining

- The subordinate object class in the containment relationship
- The superior object class in the containment relationship
- The naming attribute of the subordinate object class

The naming attribute is chosen to ensure that its value is unique for each managed object instance amongst objects that are subordinate to the same superior.

If a managed object class in a name binding is defined in a different GDMO document than the name binding, you must specify in which document the managed object class is defined.

To specify the document, prefix the managed object class name with the document name specified in the MODULE construct of the managed object class's GDMO specification—for example: "Rec. X.721 | ISO/IEC 10165-2 : 1992":system.

For information on GDMO documents, see Section 2.2.8 “Grouping GDMO Definitions Into Documents” on page 2-22

Additional Definitions in a Name Binding

In addition to a containment relationship, a name binding defines:

- Whether instances of the subordinate object class are permitted to be created and deleted by management operation
- Rules for deleting instances of the subordinate object class that contain other managed objects

A name binding can also optionally provide information for the creation of an object under a superior object, such as:

- The identity of a reference object from which attribute values for initializing the new object are obtained
- An instruction that the name of the new object is automatically assigned

Example Name Binding Definition

CODE EXAMPLE 2-6 shows the definition of the `satellite-system` name binding.

CODE EXAMPLE 2-6 GDMO Definition of the `satellite-system` Name Binding

```
...
satellite-system NAME BINDING
  SUBORDINATE OBJECT CLASS satellite;
  NAMED BY
  SUPERIOR OBJECT CLASS "Rec. X.721 | ISO/IEC 10165-2 : 1992":
    system;
  WITH ATTRIBUTE satelliteId;
  BEHAVIOUR satellite-systemBehaviour BEHAVIOUR DEFINED AS
    !For the test agent, local instances of the satellites
    will be created under the system branch of the tree
    !;
  ;
  CREATE;
  DELETE ONLY-IF-NO-CONTAINED-OBJECTS;
  REGISTERED AS { satman-binding 1 };
...
```

This example specifies that:

- Instances of the `satellite` class can be contained by the class named `system` defined in ITU-T X.721/ISO-10165-2 *Definition of Management Information*.
- The naming attribute of `satellite` instances is `satelliteId`.
- Instances of the `satellite` class can be created under an instance of `system` by management operation.
- An instance of the `satellite` class can be deleted only if it contains no other objects.

Definition of Multiple Levels of Containment

A single name binding defines only one level of containment in the MIT. To define multiple levels, multiple name bindings are required. For example, consider the object model of the `satellite` example, which defines containment relationships between managed objects as shown in FIGURE 2-4.

CODE EXAMPLE 2-7 shows how the example MIT given in FIGURE 2-4 is expressed in GDMO notation. The root-system name binding is not shown here because it is defined in an ITU-T standard GDMO definition.

CODE EXAMPLE 2-7 GDMO Definition of the Example MIT

```
...
satellite-system NAME BINDING
  SUBORDINATE OBJECT CLASS satellite;
  NAMED BY
    SUPERIOR OBJECT CLASS "Rec. X.721 | ISO/IEC 10165-2 : 1992":
      system;
  WITH ATTRIBUTE satelliteId;
  BEHAVIOUR satellite-systemBehaviour BEHAVIOUR DEFINED AS
    !For the test agent, local instances of the satellites
      will be created under the system branch of the tree
    !;
  ;
  CREATE;
  DELETE ONLY-IF-NO-CONTAINED-OBJECTS;
  REGISTERED AS { satman-binding 1 };

channel-satellite NAME BINDING
  SUBORDINATE OBJECT CLASS channel;
  NAMED BY
    SUPERIOR OBJECT CLASS satellite;
  WITH ATTRIBUTE channelId;
  BEHAVIOUR channel-satelliteBehaviour BEHAVIOUR DEFINED AS
    ! channel objects will always be contained in a
      satellite object
    !;
  ;
  CREATE;
  DELETE ONLY-IF-NO-CONTAINED-OBJECTS;
  REGISTERED AS { satman-binding 2 };

dish-channel NAME BINDING
  SUBORDINATE OBJECT CLASS dish;
  NAMED BY
    SUPERIOR OBJECT CLASS channel;
  WITH ATTRIBUTE dishId;
  BEHAVIOUR dish-channelBehaviour BEHAVIOUR DEFINED AS
    ! dish objects will always be contained in a channel
      object!;
  ;
```

```
CREATE;  
DELETE ONLY-IF-NO-CONTAINED-OBJECTS;  
REGISTERED AS { satman-binding 3 };  
...
```

2.2.6.2 Names of Managed Object Instances

Managed object instances are identified by:

- Relative distinguished names
- Fully distinguished names
- Local distinguished names

Relative Distinguished Names

The relative distinguished name (RDN) of a managed object instance represents the location of the instance in the containment tree relative to its superior object instance. The naming attribute and its value provide the RDN of an object instance. The RDN is expressed in an attribute value assertion (AVA) as *namingAttribute = "value"*.

For example, if the naming attribute of a managed object class is `satelliteId` and its value for an instance is `NorthernLights`, the RDN of the instance is `satelliteId="NorthernLights"`.

Fully Distinguished Names

The fully distinguished name (FDN) of a managed object instance represents its unique location in the containment tree. The FDN is a concatenation of the sequence of RDNs from the root of the containment tree to the instance.

For example, consider a dish contained in a channel, which is in turn contained in a satellite. The satellite is contained in an instance of the `system` object, which is below the root of the containment tree.

The root of the containment tree is represented by a forward slash (/). The RDNs of the managed object instances are as follows:

- `systemId="starless"` (for the `system` object)
- `satelliteId="NorthernLights"` (for the satellite)
- `channelId="HBO"` (for the channel)
- `dishId="Tic"` (for the dish)

The FDN of the dish is

```
/systemId="starless"/satelliteId="NorthernLights"/channelId="HBO"/dishId="Tic".
```

The derivation of this FDN is shown in FIGURE 2-5.

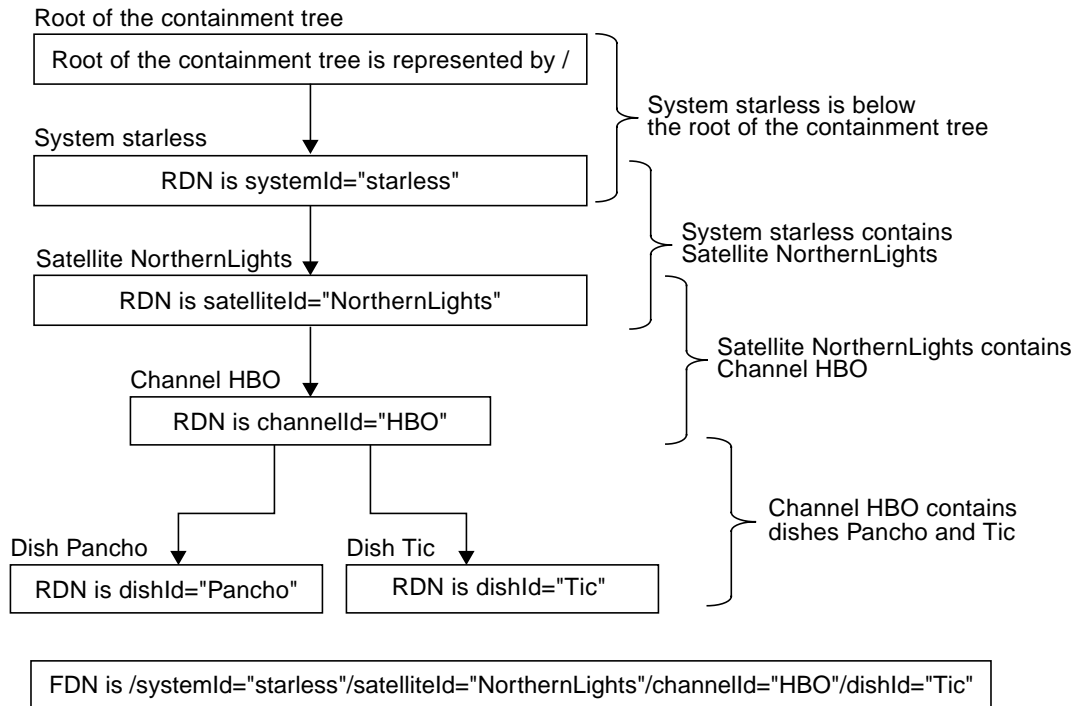


FIGURE 2-5 Containment Tree and Object Naming

Local Distinguished Names

A local distinguished name (LDN) enables agents that cannot interpret FDNs to locate managed objects. The LDN of a managed object instance represents the location of the instance in the containment tree relative to a local root. A local root is any instance in the containment tree other than the root of the containment tree. Only one local root is allowed in a containment tree.

The LDN is concatenation of the sequence of RDNs from the local root to the instance. An LDN does not begin with a forward slash (/), unlike an FDN, which always begins with a forward slash.

For example, if the satellite named `NorthernLights` in the example in FIGURE 2-5 is defined as a local root, the LDN of the dish named `Tic` is `channelId="HBO"/dishId="Tic"`.

2.2.6.3 Brace Notation for Relative and Fully Distinguished Names

Some Solstice EM tools require you to specify the FDN of a managed object instance by using brace notation instead of the slash notation described in Section 2.2.6.2 “Names of Managed Object Instances” on page 2-18. For example, the MIS Objects tool and the nickname service require you to use brace notation to specify FDNs.

In brace notation, each RDN is expressed as `{{namingAttribute, value}}`. A comma separates *namingAttribute* and *value*. If *value* is a string type (for example, an octet string) it must be enclosed in double quotes. The RDN is contained inside two pairs of braces. For example, if the naming attribute of an object class is `satelliteId` and its value for an instance is `NorthernLights`, the RDN of the instance in brace notation is `{{satelliteId, "NorthernLights"}}`.

To form the FDN, the RDNs are concatenated in a list as follows:

`{ 1stRDN, 2ndRDN, . . . nthRDN }`

Each RDN is separated by a comma from the RDN that follows it. The entire list is enclosed in a pair of braces.

For example, the FDN of the dish named `Tic` in the example in FIGURE 2-5 is expressed in brace notation as:

```
{ {{systemId, "starless" }}, {{satelliteId, "NorthernLights"}},  
  {{channelId, "HBO"}}, {{dishId, "Tic"}} }
```

2.2.7 Grouping Information Into Packages

GDMO notation requires you to group the data for a managed object class into packages. Each managed object class must have a minimum of one mandatory package. A package defines which attributes, actions, notifications, and behaviors characterize each instance of the managed object class. The attributes, actions, and notifications themselves are defined separately.

However, the package definition does specify which management operations are permitted for an attribute when it is present in the managed object class. Depending on the operations permitted, attributes can be read to recover information about the associated resource, modified to alter the current state of the associated resource, or both.

It is also possible to define conditional packages, which are present or absent depending on the characteristics of the resource being managed. The conditions under which this package is present or absent form part of the managed object class definition.

Using conditional packages makes managed object classes more flexible. Every instance of a managed object class must contain all the mandatory packages, but by adding or removing conditional packages you can allow for variations without creating a new managed object class for every case.

CODE EXAMPLE 2-8 shows the specification of the `dishPackage` package, which is contained in the `dish` managed object class.

CODE EXAMPLE 2-8 GDMO Definition of the `dishPackage` Package

```
...
dishPackage PACKAGE
    BEHAVIOUR dishPackageDefinition BEHAVIOUR DEFINED AS
        !This managed object contains the attributes for
        monitoring the customers
        !;
    ;
    ATTRIBUTES
        dishId      GET,
        censureButton
            DEFAULT VALUE WATTA-DISH-ASN1.defaultButton
            REPLACE,
        coordinates      GET-REPLACE,
        vchipId GET
    ;
    NOTIFICATIONS
        "Rec. X.721 | ISO/IEC 10165-2 : 1992":
            objectCreation,
        "Rec. X.721 | ISO/IEC 10165-2 : 1992":
            objectDeletion,
        "Rec. X.721 | ISO/IEC 10165-2 : 1992":
            attributeValueChange;
...

```

This example specifies that:

- The behavior of the `dish` managed object class is to contain attributes for monitoring customers who are using dishes.
- The following attributes are present in the `dish` managed object class:
 - `dishId`
 - `censureButton`
 - `coordinates`
 - `vchipId`
- The `dish` managed object class supports the following notifications defined in ITU-T X.721/ISO-10165-2 *Definition of Management Information*:

- objectCreation
- objectDeletion
- attributeValueChange

2.2.8 Grouping GDMO Definitions Into Documents

A GDMO document is a grouping of GDMO definitions applicable to a particular aspect of network management. Solstice EM requires that every GDMO definition is grouped into a GDMO document. You cannot insert anything other than blank characters and comments outside of a GDMO document.

The format of a GDMO document is as follows:

```
MODULE "document"
definitions
END
```

Where:

- *document* is the name of the GDMO document. It is a text string, starting with an upper-case letter. Every GDMO document must have a name. Assigning a name to a GDMO document enables EM to differentiate between items of the same name that are defined in more than one GDMO document.
- *definitions* are the GDMO definitions grouped into the GDMO document.

The `MODULE` and `END` keywords are mandatory. The `MODULE` keyword marks the start of the GDMO document. The `END` keyword marks the end of the GDMO document.

Note – The `MODULE` and `END` keywords are specific to Solstice EM. They are not part of the standard ITU-T X.722/ISO-10165-4 *Guidelines for the Definition of Managed Objects (GDMO)*.

CODE EXAMPLE 2-9 shows how the GDMO document of the satellite example is named.

CODE EXAMPLE 2-9 Naming a GDMO Document

```
MODULE "Satellite Manager"
...
END
```

In this example, the GDMO document is named `Satellite Manager`.

2.3 Abstract Syntax Notation #1 (ASN.1)

In the Solstice EM environment, all information in management requests must be represented in a machine-independent format for transmission over a network. The format used is defined in ITU-T X.208/ISO-8824 *Specification of Abstract Syntax Notation One (ASN.1)*. This standard defines a canonical form for representing all open systems interconnection (OSI) application data.

When using Solstice EM to develop applications, you need to write ASN.1 syntax definitions for all the custom data types in the GDMO definitions of your managed objects.

2.3.1 Grouping ASN.1 Syntax Definitions Into Modules

All ASN.1 syntax definitions must be grouped together into modules. You cannot insert anything other than blank characters and comments outside a module definition. There are no rules about module length or content. You are free to group definitions together in an appropriate way for your application.

The format of a module is as follows:

```
moduleRef { oid }  
DEFINITIONS ::=   
BEGIN  
definitions  
END
```

Where:

- *moduleRef* is the module reference, or name, of the module. It is a text string, starting with an upper-case letter. Every module must have a module reference.
- *oid* is the object identifier (OID) of the module. The OID provides a universally unique identifier for the module, allowing other modules to refer to it. For more information on OIDs, refer to Section 2.4 “Assigning Unique Identifiers” on page 2-30.
- *definitions* are the ASN.1 definitions grouped into the module.

The `DEFINITIONS` and `BEGIN` keywords are mandatory, and mark the start of the module. The `END` keyword is mandatory and marks the end of the module.

The beginning and end of the satellite example ASN.1 module are shown in CODE EXAMPLE 2-10.

CODE EXAMPLE 2-10 Beginning and End of an ASN.1 Module

```
SAT-MAN-ASN1 { iso(1) org(3) dod(6) internet(1) private(4)
               enterprises(1) sun(42) products(2) satman(55) }
DEFINITIONS ::=
.
.
.
END
```

In this example, the module reference is SAT-MAN-ASN1 and the OID of the module is { 1 3 6 1 4 1 42 2 55 }.

2.3.2 Defining ASN.1 Types

An ASN.1 Type is a data type defined using the ASN.1 notation.

The syntax of an ASN.1 type definition is as follows:

```
typeRef ::= type
```

Where:

- *typeRef* is the type reference, or name, of the type you want to define.
- *type* is the type reference of an existing ASN.1 type.

2.3.2.1 Type Reference

Every type must have type reference. The type references of ASN.1 universal types are always written in upper case, for example INTEGER, or BOOLEAN. For information on ASN.1 universal types, refer to Section 2.3.2.3 “Universal Types” on page 2-26.

The type references of all other ASN.1 types must start with an upper-case letter, for example NewType.

2.3.2.2 Type

A new type must always be defined using an existing type. The existing type can be either an ASN.1 universal type, or another custom type. If the custom type is defined in a different module, you have to import its definition. For more information, see Section 2.3.4 “Reusing Definitions From Other ASN.1 Modules” on page 2-28.

Definition in Terms of an ASN.1 Universal Type

CODE EXAMPLE 2-11 shows the definition of an ASN.1 type in terms of an ASN.1 universal type.

CODE EXAMPLE 2-11 Definition of the CurrentLogSize ASN.1 Type

```
CurrentLogSize ::= INTEGER
```

In this example, the CurrentLogSize type is defined to be of the INTEGER ASN.1 universal type.

Definition in Terms of Another Custom Type

A new type does not need to be defined using a universal type directly, but can be defined using another custom type. CODE EXAMPLE 2-12 shows the definition of an ASN.1 type in terms of another custom type.

CODE EXAMPLE 2-12 Definition of the SatelliteData ASN.1 Type

```
SatelliteData ::= SET OF SatelliteSeq
```

In this example, the SatelliteData type from the satellite example has been defined as a SET OF type, containing instances of the SatelliteSeq type. The definition of the SatelliteSeq type is shown in CODE EXAMPLE 2-13.

CODE EXAMPLE 2-13 Definition of the SatelliteSeq ASN.1 Type

```
SatelliteSeq ::= SEQUENCE {  
    name      GraphicString,  
    value     Integer32,  
    checkSum  CheckSum  
}
```

In this example, the `SatelliteSeq` type is composed of the values of `name`, `value`, and `checksum` in that order. The `name`, `value`, and `checksum` types are defined in other ASN.1 module specifications.

2.3.2.3 Universal Types

The ASN.1 types defined in ITU-T X.208/ISO-8824 *Specification of Abstract Syntax Notation One (ASN.1)* are referred to as universal, or built-in types. These are the building-blocks of ASN.1. All new ASN.1 types must be defined either directly or indirectly in terms of these universal types. TABLE 2-5 lists ASN.1 universal types.

TABLE 2-5 ASN.1 Universal Types

Type	Allowed Value
Simple Types	
BOOLEAN	One of TRUE or FALSE
INTEGER	A positive or negative whole number, can include zero
ENUMERATED	A defined set of values
REAL	Members of the set of real numbers
BIT STRING	An ordered sequence of zero or more bits.
OCTET STRING	An ordered sequence of zero or more octets, where each octet is an ordered sequence of eight bits
NULL	A single value, also called null
OBJECT IDENTIFIER	An OID
Structured Types	
SEQUENCE	A fixed, ordered list of types
SET	A fixed list of types, where order is not significant
SEQUENCE OF	An ordered list of zero or more values of the same type
SET OF	A list of zero or more values of the same type, where order is not significant
Other Types	
CHOICE	A fixed, unordered list of types, where the value of the new type is the value of one of the component types
ANY	A choice type whose component types are unspecified

2.3.2.4 Ranges of Allowed Values

If you need to limit the values of an ASN.1 type, you can specify a range of allowed values for the type. To specify a range, append the range in parentheses to the ASN.1 type definition. The upper and lower ends of the range must be separated by two periods (..).

CODE EXAMPLE 2-14 shows how a range of allowed values is specified for the `Integer8` ASN.1 type. The `Integer8` ASN.1 type is defined in the OMNIPoint 1 Definitions module (`/opt/SUNWconn/em/etc/asn1/vol4.asn1`).

CODE EXAMPLE 2-14 Specifying a Range of Allowed Values for an ASN.1 Type

```
Integer8 ::= INTEGER (0..255)
```

In this example the `Integer8` type is defined as an `INTEGER` type. Values of the `Integer8` type must be in the range 0 to 255.

2.3.3 Defining ASN.1 Values

An ASN.1 value is an instance of an ASN.1 type to which a value has been assigned. Define an ASN.1 value to simplify the assignment of OIDs or to specify a default value for all instances of an ASN.1 type.

The syntax of an ASN.1 value definition is as follows:

```
valueRef type ::= value
```

Where:

- *valueRef* is the value reference, or name, of the instance you want to define. To distinguish it from a type reference, a value reference starts with a lower case letter, for example, `valueName`.
- *type* is the type reference of an existing ASN.1 type.
- *value* is the value you want to assign to the instance.

CODE EXAMPLE 2-15 shows the definition of an ASN.1 value.

CODE EXAMPLE 2-15 Defining an ASN.1 Value

```
satman OBJECT IDENTIFIER ::= { 1 3 6 1 4 1 42 2 55 }
```

In this example, an instance named `satman` of type `OBJECT IDENTIFIER` is assigned the value `{ 1 3 6 1 4 1 42 2 55 }`.

2.3.4 Reusing Definitions From Other ASN.1 Modules

You can define new types or values using definitions from other ASN.1 modules, provided that these definitions have been imported. You can reuse definitions from standard modules or modules you have written yourself.

To reuse a definition from another ASN.1 module, import it into your ASN.1 module. To make a definition available to other ASN.1 modules, export it.

Note – The sections for importing and exporting definitions must come *before* ASN.1 type and value definitions in an ASN.1 module.

2.3.4.1 Importing a Definition

To import a definition, use the `IMPORTS` keyword immediately after the `BEGIN` keyword. The syntax for importing a definition is as follows:

```
BEGIN

    IMPORTS

        definitionRef
    FROM moduleRef { oid };
```

Where:

- *definitionRef* is the type or value reference of the ASN.1 definition you want to import.
- *moduleRef* is the module reference of the module that contains the definition you want to import.
- *oid* is the OID of the module from which you want to import the definition.

To import more than definition from the same ASN.1 module, list the definitions, separated by commas, before the `FROM` keyword. You can mix type and value definitions from the same module.

To import from several modules, repeat the `FROM` keyword before each module reference.

The `IMPORTS` section of the satellite example ASN.1 is shown in CODE EXAMPLE 2-16.

CODE EXAMPLE 2-16 Importing ASN.1 Definitions

```
IMPORTS
    SimpleNameType FROM
        Attribute-ASN1Module {
            joint-iso-ccitt ms(9) smi(3) part2(2) asn1Module(2) 1}
    Integer32 FROM SYNTAX-1 { iso 3 14 2 2 0 1 };
```

In this example, the `SimpleNameType` definition is imported from the `Attribute-ASN1Module` module and the `Integer32` definition is imported from the `SYNTAX-1` module.

2.3.4.2 Exporting a Definition

Exporting a definition makes it available to other ASN.1 modules. You can import a definition only if it has been exported from the module in which it is defined. By default, all definitions are exported.

If you want to restrict the list of definitions you export, include the `EXPORTS` keyword after `BEGIN`. If you include the `EXPORTS` keyword, only the definitions you list will be exported. To export no definitions from a module, include the `EXPORTS` keyword without providing a list of definitions.

The ASN.1 module for the satellite example does not include the `EXPORTS` keyword. Therefore, all definitions are exported.

The ASN.1 module for the ITU-T X.227/ISO-8650 *Connection-Oriented Protocol Specification for the Association Control Service Element* standard uses the `EXPORTS` keyword to export definitions as shown in CODE EXAMPLE 2-17.

CODE EXAMPLE 2-17 Exporting ASN.1 Definitions

```
BEGIN
EXPORTS
    acse-as-id, ACSE-apdu,
    aCSE-id, Application-context-name,
    AP-title, AE-qualifier,
    AE-title, AP-invocation-identifier,
    AE-invocation-identifier,
    Mechanism-name, Authentication-value;
```

2.4 Assigning Unique Identifiers

When you are exchanging data between different components of a managed system, you need a means of uniquely identifying items represented in the data. To identify an item uniquely, assign it an object identifier (OID).

2.4.1 Registering an OID

To ensure that an OID is globally unique, register it. A registered OID provides a globally unique identifier for each of the following:

- Managed object classes
- Attributes
- Actions
- Notifications
- Packages
- Name Bindings
- Parameters
- Attribute Groups
- Behaviors
- ASN.1 Modules

Registered OIDs are organized in a hierarchy known as the ISO registration tree. The ISO registration tree contains nodes labeled using nonnegative integer values and a text label. The top of this tree is called the root. There are three labeled nodes under root which are administered by the ISO and CCITT standards organizations. There are in turn sub-nodes under the three ISO and CCITT nodes. They are administered by ISO, CCITT, and other organizations

The OID of a node in the tree is a unique label formed by concatenating the labels of each node in the tree from root to the node. For example the OID for Solstice EM is {1 3 6 1 4 1 42 2 2 2}. FIGURE 2-6 shows how this OID is arrived at.

To ensure that your OIDs are unique, apply to the relevant authority in your own country, typically the national standards body, to be allocated your own subtree of the ISO registration tree.

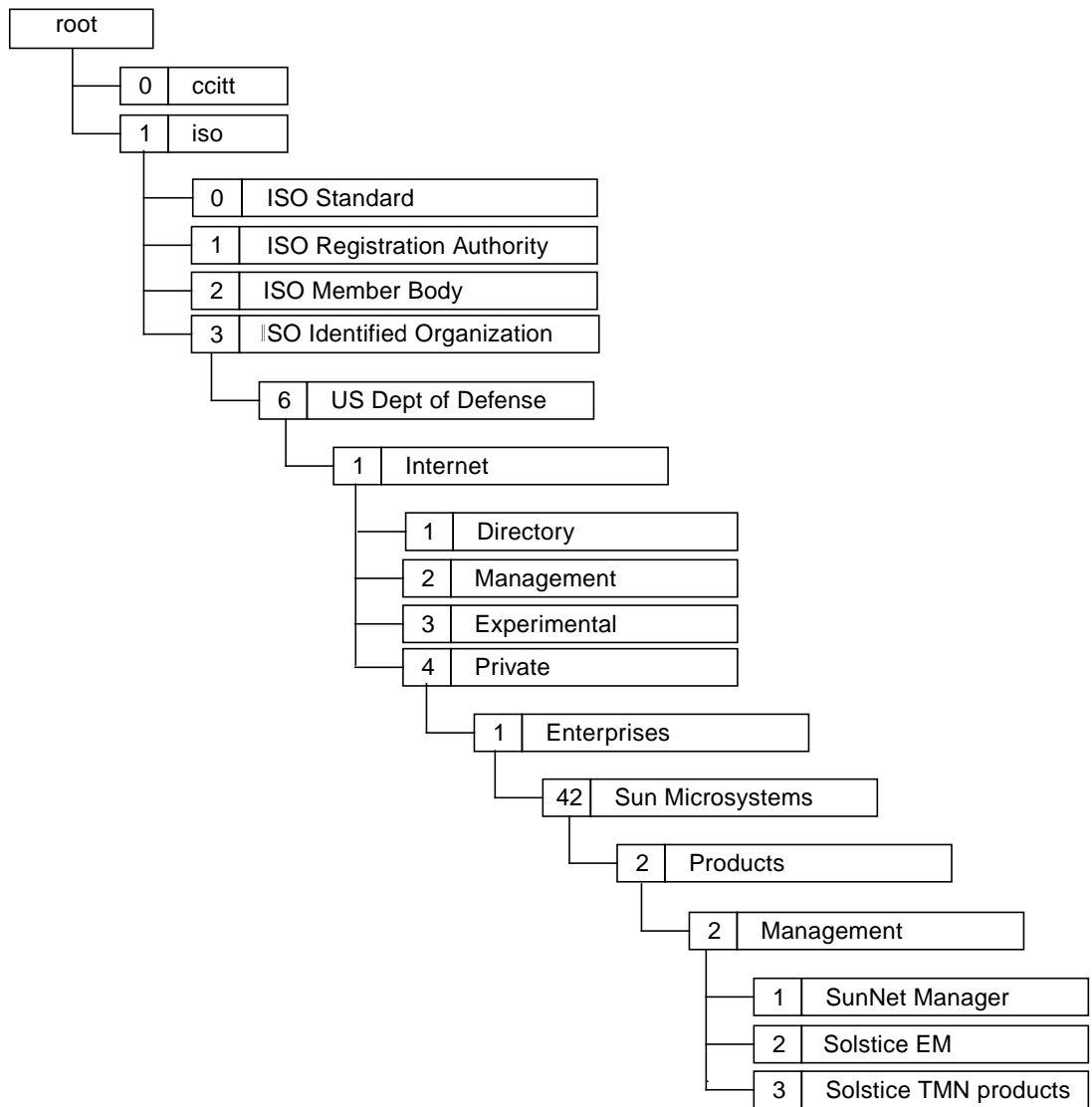


FIGURE 2-6 ISO Registration Tree

2.4.2

Guidelines for Allocating Your Own OIDs

When you have been allocated a subtree of the ISO registration tree, define a convention to ensure that every item in the subtree has a unique object identifier.

To simplify debugging, put items of the same type in their own branch of the subtree. For example, allocate one branch to object classes, another to name bindings and a third to attributes. When you use a debugging tool such as `em_debug`, which identifies items only by their OIDs, it is easier to distinguish between object classes, name bindings, and attributes if they are in separate subtrees.

The OIDs of branches of the subtree for the satellite example are shown in CODE EXAMPLE 2-18. These OIDs are assigned in the ASN.1 module specification for the satellite example.

CODE EXAMPLE 2-18 OIDs for Branches of the Subtree in the Satellite Example

```
satman OBJECT IDENTIFIER ::= { 1 3 6 1 4 1 42 2 55}
satman-objectClass OBJECT IDENTIFIER ::= {satman 3}
satman-package OBJECT IDENTIFIER ::= {satman 4}
satman-binding OBJECT IDENTIFIER ::= {satman 6}
satman-attribute OBJECT IDENTIFIER ::= {satman 7}
```

In this example, OIDs are defined for branches of the subtree containing object classes, packages, name bindings, and attributes. These OIDs are used as the roots of OIDs for individual classes, name bindings and attributes as shown in CODE EXAMPLE 2-19. These OIDs are assigned in the GDMO definition of the managed objects for satellite example.

CODE EXAMPLE 2-19 OIDs for the Satellite Example

```
satellite MANAGED OBJECT CLASS
...
    REGISTERED AS { satman-objectClass 1 };

channel MANAGED OBJECT CLASS
...
    REGISTERED AS {satman-objectClass 2};
...
satellite-system NAME BINDING
...
    REGISTERED AS { satman-binding 1 };
...
```

```
satelliteId ATTRIBUTE
...
    REGISTERED AS { satman-attribute 1 };

activeChannels ATTRIBUTE
...
    REGISTERED AS { satman-attribute 2 };
...
```

2.4.3 Notation for OIDs

You must specify OIDs in your GDMO definitions and ASN.1 module specifications by using one of the following notation types:

- Dot notation
- Brace notation

To simplify the assignment of OIDs for items under a common node, you can assign a label to the OID of the node and use the label in assigning OIDs to items under the node.

2.4.3.1 Dot Notation

Dot notation uses the integer label for each node in the ISO registration tree from root to the node of interest, separated by periods. Dot notation has the advantage of brevity, but it can be difficult to read because it contains only integers.

For example, the OID for Solstice EM is written as 1.3.6.1.4.1.42.2.2.2 in dot notation.

2.4.3.2 Brace Notation

In brace notation, each node in the registration tree is identified by one of the following:

- The text label followed by the integer label in parentheses
- The text label alone
- The integer label alone

The OID is formed by concatenating the node identifiers, with each identifier separated by white space. The OID is enclosed in braces.

For example, the OID for Solstice EM can be written in brace notation as any of the following:

- {1 3 6 1 4 1 42 2 2 2}
- {iso(1) org(3) dod(6) internet(1) private(4) enterprises(1) sun(42) products(2) management(2) em(2)}
- {iso org dod internet private enterprises sun products management em}
- {iso org dod internet private enterprises sun(42) 2 2 2}

2.4.3.3 OID Labels

An OID label is a text string that identifies an OID. Labelling an OID simplifies the assignment of OIDs for items under a common node. After you label the OID of a node, you can use the label in assigning OIDs to items under the node.

Note – OID labels are permitted only in brace notation.

CODE EXAMPLE 2-20 shows the assignment of a label to the OID of the satellite example.

CODE EXAMPLE 2-20 Labelling an OID

```
satman OBJECT IDENTIFIER ::= { 1 3 6 1 4 1 42 2 55}
```

In this example, the label `satman` is assigned to the OID of the satellite example. This OID is expressed in brace notation. CODE EXAMPLE 2-21 shows how the `satman` label is used in the assignment of an OID to a node under the `satman` node.

CODE EXAMPLE 2-21 Using an OID Label in an OID Assignment

```
satman-objectClass OBJECT IDENTIFIER ::= {satman 3}
```

2.5 Obtaining GDMO and ASN.1 Specifications for Objects

To simplify the task of writing the object model, use existing specifications as the basis of your object model whenever possible. Depending on the resources you want to manage, consider using the following as the basis of your object model:

- Existing GDMO definitions
- SNMP MIBs

2.5.1 Existing GDMO Definitions

Standards bodies such as the International Telecommunication Union - Telecommunication Standardization Sector (ITU-T) have written GDMO definitions for managed objects common in network management. Before defining your own managed objects, consult published network management standards to see if the managed objects you require have already been defined.

If no suitable standard exists, but your object model is similar to an existing object model, you can use the existing object model as the basis of your new object model. Modify the GDMO definition of the existing object model by:

- Replacing the name of the document in the `MODULE` construct with the name of the new model
- Changing the names of managed object classes and attributes to reflect those you have identified for the new model
- Adding identifiers of GDMO documents to the names of items defined in other GDMO documents
- Changing the name binding definitions to define the containment tree you require
- Adding definitions of new attributes
- Changing OIDs to those allocated for your project

Note – If you obtain GDMO definitions and ASN.1 module specifications in a single file, put the GDMO definitions in a separate file from the ASN.1 specifications before you make your object model available to Solstice EM. Solstice EM does not accept a mixture of GDMO definitions and ASN.1 module in a single file.

2.5.2 SNMP MIBs

If you are managing SNMP agents, the object model of the agent will be expressed as an SNMP management information base (MIB). To manage an SNMP agent in the Solstice EM environment, the MIB of the agent must be represented as a GDMO document and an ASN.1 module specification.

To generate a representation of an SNMP MIB as a GDMO document and an ASN.1 module specification, use either of the following:

- Concise MIB compiler (`em_cmib2gdm`)
- Load Data Definitions tool

The concise MIB compiler and the Load Data Definitions tool take an SNMP MIB as input and generates a GDMO document and an ASN.1 module specification suitable for loading into Solstice EM.

For information on how to use the concise MIB compiler and the Load Data Definitions tool, refer to the *Management Information Server (MIS) Guide*.

2.6 Making Your Object Model Available to Solstice EM

Making your object model available to Solstice EM provides the Solstice EM platform with information it requires to perform management operations on your managed objects. Making your object model available to Solstice EM involves:

- Loading your object model into the metadata repository (MDR)
- (Optional) setting agent role behavior of Solstice EM

2.6.1 Loading Your Object Model Into the MDR

In the Solstice EM environment, definitions of managed objects are stored in a metadata repository (MDR). To make your object model available to the Solstice EM environment, you must load it into the MDR.

Loading your object model into the MDR involves:

- Loading the GDMO specification by using the GDMO compiler (`em_gdm`)
- Loading the ASN.1 module specification by using the ASN.1 compiler (`em_asn1`)

For information on how to use the GDMO and ASN.1 compilers, refer to the *Management Information Server (MIS) Guide*.

2.6.2 Setting Agent Role Behavior of Solstice EM

Setting agent role behavior of Solstice EM specifies how Solstice EM maintains managed objects. Managed objects are typically maintained by an agent. However, if you do not have a dedicated agent, you can use Solstice EM in an agent role to maintain managed objects. If you are using Solstice EM in an agent role, you must set its agent role behavior.

If your managed objects are maintained by an agent, all you need to do is load your object model into the MDR. You do not need to set agent role behavior.

Before you set agent role behavior of Solstice EM, make sure you have loaded your object model into the MDR.

Depending on your requirements, you can set:

- **Default agent role behavior.** The Solstice EM management information server (MIS) provides default agent role behavior based on the definitions in your object model. Setting default agent role behavior provides the MIS with information it requires about your object model.
- **Custom agent role behavior.** If you want define your own agent role behavior, use object development tools (ODT).

Setting default agent role behavior of Solstice EM involves:

- Loading managed object class definitions into the MIS
- Loading name bindings

Solstice EM also enables you to set default agent role behavior of Solstice EM in a single operation.

Note – The procedures in the following subsections explain how to set default agent role behavior from the command line. If you want to set default agent role behavior interactively, use the Load Data Definitions tool, as explained in the *Management Information Server (MIS) Guide*.

2.6.2.1 Loading Managed Object Class Definitions Into the MIS

Loading managed object class definitions into the MIS provides the MIS with information it requires to perform management operations on instances of the managed object classes.

How to load managed object class definitions into the MIS depends on whether you want your managed object class definitions to be volatile or persistent.

- If you want your definitions to be volatile, use `em_compose_oc`
- If you want your definitions to be persistent, use `em_compose_poc`

For information on how to use `em_compose_oc` and `em_compose_poc`, refer to the *Management Information Server (MIS) Guide*.

2.6.2.2 Loading Name Bindings

Loading name bindings provides Solstice EM with the information it requires to create and locate managed objects in the MIT. To load name bindings, use the `em_load_name_bindings` command.

For information on how to use the `em_load_name_bindings` command, refer to the *Management Information Server (MIS) Guide*.

2.6.2.3 Setting Default Agent Role Behavior in a Single Operation

If you want your managed object class definitions to be persistent, you can set default agent role behavior of Solstice EM in a single operation. To set default agent role behavior of Solstice EM in a single operation, use the `em_compose_all` script. This script is a wrapper that executes `em_compose_poc` and `em_load_name_bindings`.

For information on how to use the `em_compose_all` script, refer to the *Management Information Server (MIS) Guide*.

Enabling Applications to Access Managed Objects

To manage a network, the applications you develop must have access to current data about managed resources. In the Solstice EM environment, managed resources are represented as managed objects. Access to managed objects is provided by a management information server (MIS). The MIS is a repository for all management data and functions. It makes information about managed objects available to its clients, both applications and services. Your applications access the MIS by connecting to it through an application programming interface (API) called the Portable Management Interface (PMI). The MIS then makes information about managed objects available to your applications and their users. For more information about the MIS, refer to the *Management Information Server (MIS) Guide*.

This chapter explains how to enable your management applications to access managed objects stored in an MIS.

- Section 3.1 “Connecting to an MIS” on page 3-1
- Section 3.2 “Disconnecting From an MIS” on page 3-4
- Section 3.3 “Bypassing the MIS to Access Solstice EM Databases” on page 3-5

3.1 Connecting to an MIS

In the Solstice EM environment, managed objects are accessed through an MIS. To gain access to managed objects, an application must connect to the MIS.

To enable an application to connect to an MIS, use the `Platform` class of the Portable Management Interface (PMI). An instance of the `Platform` class represents a connection from an application to an MIS, and contains all the information about the MIS that the application requires.

Enabling an application to connect to an MIS involves:

- Creating and initializing an instance of the `Platform` class
- Calling the `connect` function of the `Platform` class

3.1.1 Creating and Initializing an Instance of the Platform Class

To connect to an MIS, an application must instantiate and initialize an instance of the `Platform` class. When you call the constructor of the `Platform` class, you must specify the platform type.

Note – The only supported platform type is `duEM`.

Code for creating and initializing an instance of the `Platform` class is shown in CODE EXAMPLE 3-1.

CODE EXAMPLE 3-1 Creating and Initializing a Platform Instance

```
...
#include <pmi/hi.hh>           // High Level PMI
...
Platform plat;               // Create instance
...
plat = Platform(duEM);       // Initialize instance
...
```

Note – The constructor of the `Platform` class does not throw an exception if the attempt to initialize the instance of `Platform` fails. If you want to check for errors during initialization, use the `get_error_type` function as described in Section 4.1.2 “Using the `get_error_type` Function” on page 4-2.

3.1.2 Calling the connect Function of the Platform Class

After creating and initializing an instance of the `Platform` class, you need to call the `connect` function of the `Platform` class to establish a connection between your application and an MIS. In the call to the `connect` function, you must specify:

- **The location of the MIS.** This location is the host name of the machine on which the MIS is running.
- **The name of your application.** When a user runs your application, the Access Control module verifies that the user has permission to access the MIS by using that application. For more information on access control, refer to Chapter 12.

Code for connecting to an MIS and checking for connection errors is shown in CODE EXAMPLE 3-2.

CODE EXAMPLE 3-2 Calling the `connect` Function

```
...
#include <pmi/hi.hh>           // High Level PMI
...
if (!plat.connect("a101", "simplemanager"))
{
    cout << "Failed to connect to " << "a101" << endl;
    cout << plat.get_error_string() << endl;
}
...
```

In this example, the location of the MIS is specified by the host name `a101`. The name of the application is `simplemanager`. If the connection fails, the error message returned by `get_error_string` is displayed. For more details on error handling, refer to Chapter 4.

3.2 Disconnecting From an MIS

An application is automatically disconnected when it exits, or when the MIS that it is connected to is shut down. Therefore, in most cases you do not need to provide code for disconnecting your applications from an MIS.

If you want your application to continue running after disconnecting itself from the MIS, include in the application a call to the `disconnect` function of the `Platform` class. For example, if you are developing an application that monitors a piece of equipment across a modem line and you want to make most efficient use of the modem connection, you can enable the application to:

- Connect to an MIS
- Gather information from the MIS
- Disconnect from the MIS
- Process the information it has gathered from the MIS
- Reconnect to the MIS when it needs to gather information again

If you want your application to perform a specific operation when it is disconnected from the MIS, enable your application to handle events that indicate that the application is disconnected. For information on how to enable an application to handle events, refer to Chapter 7.

Code for disconnecting from the MIS and checking for disconnection errors is shown in CODE EXAMPLE 3-3.

CODE EXAMPLE 3-3 Calling the `disconnect` Function

```
...
#include <pmi/hi.hh>           // High Level PMI
...
if (!plat.disconnect())
{
    cout << "Failed to disconnect" << endl;
}
...
```



Caution – When your application calls the `disconnect` function, instances of all PMI classes are destroyed. All metadata cached in your application is also destroyed. Make sure that your application does not reference any of the PMI instances or metadata after they have been destroyed. Any attempt to reference an instance or metadata that has been destroyed will cause your application to fail.

3.3 Bypassing the MIS to Access Solstice EM Databases

The MIS is a repository for all management data and functions. It makes data about managed objects available to applications. Managed object data are stored in a number of databases, which the MIS communicates with. Different services are associated with different databases, depending on the type of managed objects stored in a database. For example, the log service is associated with the database that stores log objects.

Normally, applications access Solstice EM databases through the MIS. You can bypass the MIS to access Solstice EM databases directly if you want to gain direct access to log record data.

Bypassing the MIS involves:

- Getting information required for a database connection
- Passing database information to a database application development tool

3.3.1 Getting Information Required for a Database Connection

To connect directly to a database, an application requires information on the database. To get the information your application needs to connect to a database, use the `EMDBConnectInfo` class. The `EMDBConnectInfo` class represents all the information about the database that an application requires.

Getting information required for a database connection involves:

- Creating and initializing an instance of the `EMDBConnectInfo` class
- Calling functions of the `EMDBConnectInfo` class

3.3.1.1 Creating and Initializing an Instance of the `EMDBConnectInfo` Class

To get the information it requires, an application must create and initialize an instance of the `EMDBConnectInfo` class. When you call the constructor of the `EMDBConnectInfo` class, you must specify:

- **The location of the database server.** This location is the host name of the machine on which the database server is running.
- **The service associated with log objects in the database.** This service must be specified as `LOG_SVC`.

Note – The only service supported is the log service.

Code for creating and initializing an instance of the `EMDBConnectInfo` class is shown in CODE EXAMPLE 3-4.

CODE EXAMPLE 3-4 Creating and Initializing an Instance of the `EMDBConnectInfo` Class

```
...
#include <dbapi/dbcon.hh>
...
    RWCString misname("indus");

    //Create the database connection object
    EMDBConnectInfo conn_info(misname,EMDBConnectInfo::LOG_SVC);
...
```

In this example, the location of the database server is specified by the host name `indus`. The service associated with log objects in the database is specified as `LOG_SVC`.

3.3.1.2 Calling Functions of the `EMDBConnectInfo` Class

Once you have created and initialized an instance of the `EMDBConnectInfo` class, call functions of the `EMDBConnectInfo` class to obtain information about the database you want to connect to. Exactly which information is needed depends on the function you call to connect to the database.

Use the functions of the `EMDBConnectInfo` class listed in TABLE 3-1 to get information you need. By using these functions to get the required information, you make your code for connecting to a database independent of the implementation of the database.

TABLE 3-1 Functions for Getting Information About a Database

Information	Function
The name of the dynamic library that Rogue Wave Software DBTools.h++ uses for interacting with the database, for example <code>librwinf.so</code> . This library depends on the database server that Solstice EM uses internally. Use the <code>get_server_type</code> function only with the Rogue Wave Software DBTools.h++ database application development tool.	<code>get_server_type</code>
The host name of the database server.	<code>get_server_name</code>
The database user name as defined by the database administrator. This is not necessarily the user's UNIX user name.	<code>get_user_name</code>
The database user's password.	<code>get_user_password</code>
The database name.	<code>get_database_name</code>
The database user's role as defined by the database administrator.	<code>get_role</code>
The status of the <code>EMDBConnectInfo</code> object. It is 0 if the object is valid and -1 if it is invalid.	<code>get_status</code>

For an example of how these functions are called, refer to CODE EXAMPLE 3-5.

3.3.2 Passing Database Information to a Database Application Development Tool

When you have obtained the information you need, pass it to a database application development tool (for example, Rogue Wave Software DBTools.h++) to connect your application to the database.

CODE EXAMPLE 3-5 shows code for getting information about a Solstice EM database and passing that information to a function for connecting to a database.

CODE EXAMPLE 3-5 Connecting to a Solstice EM Database

```
...
//Use the connection information to connect to database
RWDBDatabase aDB = RWDBManager::database (
    conn_info.get_server_type(),
    conn_info.get_server_name(),
    conn_info.get_user_name(),
    conn_info.get_user_password(),
    conn_info.get_database_name()
);
...
```

In this example, the static function `database` of the Rogue Wave Software `DBTools.h++` class `RWDBManager` is called to establish a connection between the application and a database. The database function requires the following information:

- The database server type
- The host name of the database server
- The database user name
- The database user's password
- The database name

3.3.3 Example Database Connection Program

An example program for getting information directly from the Solstice EM database is shown in CODE EXAMPLE 3-6.

CODE EXAMPLE 3-6 Getting Information Directly From a Solstice EM Database

```
// Copyright 18 Aug 1999 Sun Microsystems, Inc. All Rights Reserved.
#pragma ident  "@(#)dbapi_example.cc          1.5 99/08/18 Sun Microsystems"
//
// Purpose: To get all the log names from the database.
//
// How:      It uses DB Connection API to get connect information and
//           DBTools.h++ to retrieve the log names.
//
// Syntax: dbapi_example
//
```



```
#include <rw/cstring.h>
#include <rw/db/dbmgr.h>
#include <rw/db/db.h>
#include <dbapi/dbcon.hh>
#include <pmi/hi.hh>

main() {

    RWCString misname("indus");

    //Create the database connection object
    EMDBConnectInfo conn_info(misname,EMDBConnectInfo::LOG_SVC);

    if(conn_info.get_status() == -1)
    {
        cout << "Could not get connection information" << endl;
        exit(1);
    }

    //Use the connection information to connect to database
    RWDBDatabase aDB = RWDBManager::database (
        conn_info.get_server_type(),
        conn_info.get_server_name(),
        conn_info.get_user_name(),
        conn_info.get_user_password(),
        conn_info.get_database_name()
    );

    RWDBTable table = aDB.table("log");

    // Create the selector to get log names.
    //-----
    RWDBSelector anEmSelector = aDB.selector();
    anEmSelector << table["logid"];
    anEmSelector.where(table["logidtype"] == "string");
    RWDBReader rdr = anEmSelector.reader();

    RWCString logname;
    while(rdr())
    {
        rdr >> logname;
        cout << "Logid: " << logname << endl;
    }
}
```

In this example, the application connects directly to a database containing managed objects associated with the log service. The application then extracts from the database the identifier of each log object and displays it.

Handling Errors

Users need to know when an attempted network management operation has failed. By providing accurate information on why the operation failed, your applications can ease a user's work by indicating the corrective action required when problems occur.

This chapter explains how to handle errors in your applications.

- Section 4.1 “Testing for the Success a Function Call” on page 4-1
- Section 4.2 “Providing Error Information to Users” on page 4-3

4.1 Testing for the Success a Function Call

To provide accurate diagnostic information, test for the success of any function call that may, in some circumstances, fail to return the desired result. Solstice EM enables you to test for the success of a function call by:

- **Using the overloaded NOT operator.** Use the overloaded NOT operator only when the return value indicates if the function call achieved the desired result.
- **Using the `get_error_type` function.** Use the `get_error_type` function when the return value does not indicate if the function call achieved the desired result, for example, with constructors.

Note – Use the `get_error_type` function only for synchronous operations. To test for the success of an asynchronous operation, use the `get_except` function of the `Waiter` class as explained in Section 8.4.1 “Verifying the Result of an Asynchronous Operation” on page 8-23.

4.1.1 Using the Overloaded NOT Operator

Most PMI classes contain an overloaded NOT (!) operator to simplify error checking. The overloaded NOT operator provides a shorthand means to verify that the result of a function call is the desired one. Use the overloaded NOT operator only when the return value indicates if the function call achieved the desired result. For example, any function that returns a `Result` is suitable for use with the overloaded NOT operator.

Code for checking for errors by using the overloaded NOT operator is shown in CODE EXAMPLE 4-1.

CODE EXAMPLE 4-1 Using the Overloaded NOT Operator for Error Checking

```
...
#include <pmi/hi.hh>          // High Level PMI
...
Image image("/systemId='test'");
...
if (!image.boot()) {
    cout << image.get_error_string() << endl;
}
...
```

In this example, the overloaded NOT (!) operator acts on the call to `boot` to verify that the `Image` instance was activated. If it was not activated, the `get_error_string` function is called and the error string returned is displayed. The `Image` class inherits the `get_error_string` function from the `Error` class.

4.1.2 Using the `get_error_type` Function

For some function calls, it is not possible to use the overloaded NOT operator to verify that the result of the function call is the desired one. For example, you cannot use the overloaded NOT operator for constructors because their return values do not indicate whether the attempt to instantiate an object was successful.

In such cases, use the `get_error_type` function of the `Error` class to verify that the result of the function call is the desired one. Many of the classes of the Solstice EM APIs are derived from the `Error` class, enabling you to use the `get_error_type` function to test for the success of Solstice EM API function calls.

The `get_error_type` function returns an enumerated error type. The list of possible error types is given in the `/opt/SUNWconn/em/include/pmi/error.hh` file.

Code for checking for errors by using the `get_error_type` function is shown in CODE EXAMPLE 4-2.

CODE EXAMPLE 4-2 Using the `get_error_type` Function for Error Checking

```
...
#include <pmi/hi.hh>          // High Level PMI
...
Image im = Image("/systemId=\"test\"");
if (im.get_error_type() != PMI_SUCCESS) {
    cout << im.get_error_string() << endl;
}
...
```

In this example, the `get_error_type` function verifies that the attempt to instantiate the `Image` class is successful. If the attempt is unsuccessful, the `get_error_string` function is called and the error string returned is displayed. The `Image` class inherits the `get_error_type` and `get_error_string` functions from the `Error` class.

4.2 Providing Error Information to Users

When you check for errors, use the error-handling functions provided by Solstice EM to obtain information on the error and present that information to the user.

The error-handling functions of Solstice EM are provided by the `Error` class of the high-level PMI. Many of the classes of the Solstice EM APIs are derived from the `Error` class, enabling you to use a set of common functions for handling errors related to Solstice EM API calls. Using a common set of error-handling functions enables your applications to provide error checking that is simple and consistent for all Solstice EM API calls.

Note – Use the error-handling functions provided by the `Error` class only for synchronous operations. To obtain information on why an asynchronous operation failed, use functions of the `ExceptionType` class as explained in Section 8.4.1 “Verifying the Result of an Asynchronous Operation” on page 8-23.

CODE EXAMPLE 4-1 and CODE EXAMPLE 4-2 show how to use the `get_error_string` function of the `Error` class to display the reason a function call did not succeed.

Performing Operations on Managed Objects

An application manages a network by monitoring and controlling managed resources in the network. In the EM environment, managed resources are represented as managed objects. An application monitors and controls managed resources by performing operations on managed objects. An application performs operations to add managed objects to and remove them from the network management environment; to get information about managed objects; and to change the state of a managed object.

This chapter explains how to enable your applications to perform management operations on managed objects.

- Section 5.1 “Management Operations” on page 5-2
- Section 5.2 “Creating a Managed Object” on page 5-2
- Section 5.3 “Selecting a Managed Object” on page 5-8
- Section 5.4 “Updating an Image Instance” on page 5-14
- Section 5.5 “Deleting a Managed Object” on page 5-15
- Section 5.6 “Getting Attribute Values From an Object” on page 5-17
- Section 5.7 “Setting Attribute Values of an Object” on page 5-19
- Section 5.8 “Performing an Action on an Object” on page 5-23
- Section 5.9 “Tracking Changes to an Object” on page 5-26
- Section 5.10 “Retrieving Data From the Metadata Repository” on page 5-27
- Section 5.11 “Simulating an Agent Object” on page 5-30
- Section 5.12 “Representing MIS Instances Locally in an Application” on page 5-35

5.1 Management Operations

The Solstice EM APIs are independent of any particular management protocol or service. When you use them to develop management applications, you do not need to take account of the protocol used for communications between the agent and the manager.

The Solstice EM APIs support management operations common to most network management environments, namely:

- Creating a managed object
- Deleting a managed object
- Getting attribute values from an object
- Setting attribute values of an object
- Performing an action on an object

5.2 Creating a Managed Object

To enable users of your applications to add a managed resource to the network, an application must create a managed object to represent the managed resource. The managed resource can be a physical device such as a host, server, router, or subnet, or it can be a conceptual entity such as a line, a queue, or some other aspect of network operation that can be represented as a managed object.

An application can create an object only if the specification of the managed object allows it to be created by a management operation. Specifically, the `NAME BINDING` clause of the GDMO specification of the managed object class must contain the `CREATE` construct. Otherwise, any request to create an instance of the class is denied.

To create a managed object, use the `Image` class. An instance of the `Image` class is a local representation of a managed object. A local representation is cached within your network management application.

An instance of `Image` gives you access to the methods and attributes of a managed object. It provides attribute-like access to object and attribute schema information. Although the actual managed object is in a management information server (MIS), or in an agent in the network, you can treat an instance of `Image` as if it is the managed object itself.

Creating a managed object involves:

- Creating and initializing an instance of `Image`
- Activating the instance of `Image`

- Verifying if the managed object exists
- Initializing attributes of the managed object
- Adding the managed object to the MIS

5.2.1 Creating and Initializing an Instance of Image

To create a managed object, an application must create and initialize an instance of the `Image` class. When you call the constructor of the `Image` class, you must specify:

- The fully distinguished name (FDN), local distinguished name (LDN), or nickname of the managed object that the instance of `Image` represents
- The managed object class that the managed object is an instance of

Code for creating and initializing an instance of the `Image` class is shown in CODE EXAMPLE 5-1.

CODE EXAMPLE 5-1 Creating and Initializing an `Image` Instance

```
...
#include <pmi/hi.hh>           // High Level PMI
#include <rw/cstring.h>        // Rogue Wave RWCString
...
{
void create_channel(RWCString parent, RWCString channel_name)
    // Construct the distinguished name of the channel object
    RWCString fdn = parent + "/channelId=\"" + channel_name + "\"";

    cout << "Creating: " << fdn.data() << endl;

    Image channel_image(DU(fdn), DU("channel"));
    if (channel_image.get_error_type() != PMI_SUCCESS) {
        cout << channel_image.get_error_string() << endl;
        return;
    }
...
}
...
```

In this example, the constructor of the `Image` class is called in the body of a function named `create_channel`. The `Image` object represents an instance of a managed object class named `channel`. The FDN of the `channel` managed object instance is constructed from information passed as parameters to the `create_channel` function:

- The FDN of the managed object that contains this instance of `channel`
- The value of the naming attribute of this instance of `channel`

5.2.2 Activating the Instance of Image

Activating an `Image` instance loads attribute information into the `Image` instance from a managed object in an MIS or an agent. Until an `Image` instance is activated, it represents a potential managed object. Once the `Image` instance has been activated, it represents and contains information from an actual managed object.

To activate an `Image` instance, call the `boot` function of the `Image` class. The effect of calling the `boot` function depends on whether the managed object that the `Image` instance represents already exists:

- If the managed object does not exist, calling the `boot` function retrieves the GDMO description of the managed object from the MIS.
- If the managed object already exists, calling the `boot` function retrieves the GDMO description and the values of all the attributes of the managed object from the MIS.

The `boot` function is an overloaded function of which there are two versions. One version takes an attribute list as a parameter. The other version does not take an attribute list as a parameter. To ensure that the `boot` function loads information on all attributes defined for the managed object, call the version of `boot` that does not take an attribute list.

Note – A call to `boot` will fail if the identity of the managed object is invalid for the managed object class specified in the call to the `Image` constructor.

Code for activating an `Image` instance and checking for errors is shown in CODE EXAMPLE 5-2.

CODE EXAMPLE 5-2 Activating an `Image` Instance

```
...
#include <pmi/hi.hh>           // High Level PMI
...
    if (!channel_image.boot()) {
        cout << channel_image.get_error_string() << endl;
        return;
    }
...
```

In this example, the overloaded NOT (!) operator acts on the call to `boot` to verify that the `Image` instance was activated. If it was not activated, an error message explaining the reason for the failure is displayed. To ensure that information for all attributes is loaded, the version of `boot` that does not take an attribute list is called.

5.2.3 Verifying if the Managed Object Exists

Verify if the managed object that the `Image` instance represents exists before trying to create it. If your application tries to create a managed object that already exists, an exception will be thrown. To verify if the managed object exists, call the `exists` function of the `Image` class.

Call the `exists` function only after you have activated the `Image` instance. If the `Image` instance has not already been activated, the `exists` function returns `FALSE` even if the managed object exists.

Code for verifying if a managed object exists is shown in CODE EXAMPLE 5-3.

CODE EXAMPLE 5-3 Verifying if a Managed Object Exists

```
...
#include <pmi/hi.hh>                // High Level PMI
...
    if (channel_image.exists()) {
        cout << "Channel already exists!" << endl;
        return;
    }
...
```

In this example, the function that contains this code returns without taking any further action if the managed object exists.

5.2.4 Initializing Attributes of the Managed Object

If the managed object does not already exist, initialize its attributes. Make sure that:

- You initialize all mandatory attributes, that is all attributes that the GDMO specification of the managed object class requires to be initialized.
- The values you initialize the attributes to are allowed by the GDMO specification of the managed object class.
- You initialize only attributes that the GDMO specification of the managed object class allows to be initialized.

To initialize the attributes of the managed object, call functions of the `Image` class for setting attributes. The function you call depends on the type of the attribute as defined in the GDMO specification of the managed object class. For full details, refer to Section 5.7 “Setting Attribute Values of an Object” on page 5-19.

Check for errors each time you initialize an attribute. If a single attribute fails to be initialized, the creation of the managed object may fail. If you include error checking in each PMI call to initialize an attribute, you will know immediately why an attempt to create a managed object failed.

Code for initializing attributes of a managed object is shown in CODE EXAMPLE 5-4.

CODE EXAMPLE 5-4 Initializing Managed Object Attributes

```
...
#include <pmi/hi.hh>                // High Level PMI
...
    if (!channel_image.set_str("administrativeState","unlocked")) {
        cout << "Can't set administrativeState - ";
        cout << channel_image.get_error_string() << endl;
        return;
    }
    if (!channel_image.set_str("operationalState","enabled")) {
        cout << "Can't set operationalState - ";
        cout << channel_image.get_error_string() << endl;
        return;
    }
    if (!channel_image.set_long("transmitDelay",30)) {
        cout << "Can't set transmitDelay - ";
        cout << channel_image.get_error_string() << endl;
        return;
    }
...

```

In this example, attributes of the managed object are initialized as follows:

- The `set_str` function sets the `administrativeState` attribute to the text `unlocked`.
- The `set_str` function sets the `operationalState` attribute to the text `enabled`.
- The `set_long` function sets the `transmitDelay` attribute to the value 30.

If an attempt to set an attribute fails, an error message is displayed to indicate which attribute was not set and why.

5.2.5 Adding the Managed Object to the MIS

After initializing attributes of the managed object, add the managed object to the MIS that you have connected your application to. For information on how to connect an application to an MIS, refer to Section 3.1 “Connecting to an MIS” on page 3-1. To add a managed object to an MIS, call the `create` function of the `Image` class.

Code for adding a managed object to an MIS is shown in CODE EXAMPLE 5-5.

CODE EXAMPLE 5-5 Adding a Managed Object to an MIS

```
...
#include <pmi/hi.hh>                // High Level PMI
...
    if (!channel_image.create()) {
        cout << "Creation failed: ";
        cout << channel_image.get_error_string() << endl;
        return;
    } else {
        cout << "Creation succeeded!" << endl;
    }
...

```

In this example, the overloaded NOT (!) operator acts on the call to `create` to verify that the managed object was created. If it was not created, an error message explaining the reason for the failure is displayed.

5.2.6 Example Object Creation Function

An example of a function that contains all the code for creating a managed object is shown in CODE EXAMPLE 5-6.

CODE EXAMPLE 5-6 Example Object Creation Function

```
...
#include <pmi/hi.hh>                // High Level PMI
#include <rw/cstring.h>             // Rogue Wave RWCString
...
void create_channel(RWCString parent, RWCString channel_name)
{
    // Construct the distinguished name of the channel object
    RWCString fdn = parent + "/channelId=\"" + channel_name + "\"";

    cout << "Creating: " << fdn.data() << endl;

    Image channel_image(DU(fdn), DU("channel"));
    if (channel_image.get_error_type() != PMI_SUCCESS) {
        cout << channel_image.get_error_string() << endl;
        return;
    }
    if (!channel_image.boot()) {

```

```

        cout << channel_image.get_error_string() << endl;
        return;
    }
    if (channel_image.exists()) {
        cout << "Channel already exists!" << endl;
        return;
    }
    if (!channel_image.set_str("administrativeState","unlocked")) {
        cout << "Can't set administrativeState - ";
        cout << channel_image.get_error_string() << endl;
        return;
    }
    if (!channel_image.set_str("operationalState","enabled")) {
        cout << "Can't set operationalState - ";
        cout << channel_image.get_error_string() << endl;
        return;
    }
    if (!channel_image.set_long("transmitDelay",30)) {
        cout << "Can't set transmitDelay - ";
        cout << channel_image.get_error_string() << endl;
        return;
    }

    if (!channel_image.create()) {
        cout << "Creation failed: ";
        cout << channel_image.get_error_string() << endl;
        return;
    } else {
        cout << "Creation succeeded!" << endl;
    }
}
...

```

5.3 Selecting a Managed Object

When you perform a management operation on an existing managed object, you have to select the managed object in one of the following ways:

- Specifying the FDN or LDN of a managed object
- Specifying the nickname of a managed object

5.3.1 Selecting a Managed Object by Specifying its FDN or LDN

To select a managed object by specifying its FDN or LDN, create and initialize an instance of the `Image` class, specifying only the FDN or LDN. There is no need to specify the managed object class.

By specifying only the FDN or LDN of the managed object, you can verify if the managed object exists by testing if the attempt to instantiate the `Image` class was successful. If the FDN or LDN does not identify an existing managed object, the call to the constructor of the `Image` class fails.

Code for selecting a managed object by specifying its FDN is shown in CODE EXAMPLE 5-7.

CODE EXAMPLE 5-7 Selecting a Managed Object by Specifying its FDN

```
...
#include <pmi/hi.hh>           // High Level PMI
#include <rw/cstring.h>        // Rogue Wave RWCString
...
Boolean delete_object(RWCString fdn)
{
    Image del_image(fdn.data());
    if (del_image.get_error_type() != PMI_SUCCESS) {
        cout << del_image.get_error_string() << endl;
        return FALSE;
    }
    ...
}
...
```

In this example, the constructor of the `Image` class is called in the body of a function named `delete_object`. The FDN of the managed object is constructed from information passed as a parameter to the `delete_object` function. If the FDN passed to the constructor of the `Image` class does not identify an existing managed object, the function that contains this code returns `FALSE`.

5.3.2 Selecting a Managed Object by Specifying its Nickname

In an MIT with many levels of containment, FDNs become long and complicated, particularly the FDNs of managed objects that are many levels below the root of the MIT. To simplify the task of selecting managed objects, assign nicknames to managed objects and select managed objects by specifying their nicknames.

You are free to choose the nickname you assign to a managed object. Assigning short and memorable nicknames to managed objects reduces the possibility of coding errors and makes your code easier to read.

Selecting a managed object by specifying its nickname involves:

- Starting and configuring the nickname service
- Getting the `Image` instance associated with a nickname

In addition, the `Image` class provides functions for getting and setting the nickname of an `Image` instance.

5.3.2.1 Starting and Configuring the Nickname Service

The nickname service translates between nicknames and FDNs of managed objects. To enable your application to locate a managed object specified by its nickname, ensure that the nickname service has been started and configured before you run your application.

Starting and configuring the nickname service involves:

- Starting the nickname service daemon
- Adding the nickname service to the MIS
- Assigning nicknames to managed objects
- Loading nickname assignments into the nickname service

Starting the Nickname Service Daemon

The nickname service daemon (`em_nnmpa`) is started automatically whenever the MIS is started. To start the MIS, execute the `em_services` command. For more information, refer to the *MIS Guide*.

Adding the Nickname Service to the MIS

To enable an application to communicate with the nickname service, the nickname service must be added to the MIS. The MIS treats the nickname service as a management protocol adapter (MPA).

By default, the nickname service is added automatically to the MIS when Solstice EM is started. When the nickname service is added automatically to the MIS, the default MIS host and port number are assumed. If you want to specify a different MIS host or port number, add the nickname service to the MIS manually.

Note – The MIS must be started before you add the nickname service to the MIS manually.

To add the nickname service to the MIS manually, type:

```
hostname% em_nnadd -m MPAhost [-h MIShost] [-n port] [-help]
```

Where:

- **-m MPAhost** specifies that the nickname server is running on the host named *MPAhost*.
- **-h MIShost** specifies that the MIS host is the remote host named *MIShost*. The **-h** flag is optional. The default host is the local host.
- **-n port** specifies that the nickname service uses port number *port* on the MIS host. The **-n** flag is optional. The default is port number 5555.
- **-help** prints the usage message for the `em_nnadd` command.

Assigning Nicknames to Managed Objects

Assigning nicknames to managed objects defines the mapping between FDNs of managed objects and nicknames you want to assign to the managed objects.

To define mappings between FDNs and nicknames, create a text file that contains the mappings. In the text file, define each mapping by using a pair of lines in the following format:

```
fdn  
nickname
```

Where:

- *fdn* is the FDN of the managed object. You must express the FDN in brace notation. For more information, see Section 2.2.6.3 “Brace Notation for Relative and Fully Distinguished Names” on page 2-20.
- *nickname* is the nickname associated with the FDN. *nickname* must be a text string without quotes.

Comment lines are allowed in the file. Start each comment line with the hash character (#).

Solstice EM does not impose any restrictions on the number of mappings in a file.

An example of mappings between FDNs and nicknames is shown in CODE EXAMPLE 5-8.

CODE EXAMPLE 5-8 Mappings Between FDNs and Nicknames

```
# This is a sample em_nnconfig input file.
# Comment lines can be included. This line is a comment line.
# Entries are organized in pairs of lines.
# The first line in a pair is the FDN.
# The next line that is neither blank nor a comment is the nickname.

{ {{ systemId, "starless" }} }
starless

{ {{ systemId, "starless" }}, {{ satelliteId, "NorthernLights" }} }
NL-sat

{ {{ systemId, "starless" }}, {{ subsystemId, "EM-MIS" }}}
MIS-subsystem
```

In this example, nicknames are assigned to managed objects as follows:

- The managed object identified by the FDN `/systemId="starless"` is assigned the nickname `starless`.
- The managed object identified by the FDN `/systemId="starless"/satelliteId="NorthernLights"` is assigned the nickname `NL-sat`.
- The managed object identified by the FDN `/systemId="starless"/subsystemId="EM-MIS"` is given the nickname `MIS-subsystem`.

Loading Nickname Assignments Into the Nickname Service

Loading nickname assignments into the nickname service provides the nickname service with the information it requires to translate between FDNs and nicknames.

To load nickname assignments into the nickname service, type:

```
hostname% em_nnconfig file
```

Where *file* is the name of a file that contains the nickname assignments that you want to load into the nickname service. The required format of this file is defined in “Assigning Nicknames to Managed Objects” on page 11.

5.3.2.2 Getting the Image Instance Associated With a Nickname

To get the Image instance associated with a nickname, call the static function `find_by_nickname` of the Image class. In the call to `find_by_nickname`, specify the nickname of the managed object you want to select. The `find_by_nickname` function returns the Image instance that represents the managed object you want to select. When you have obtained the Image instance, you can perform management operations by calling functions on this Image instance.

Code for getting the Image instance associated with a nickname is shown in CODE EXAMPLE 5-9.

CODE EXAMPLE 5-9 Getting the Image Instance Associated With a Nickname

```
...
#include <pmi/hi.hh>                // High Level PMI
...
CDU nicnam = "NL-sat" ;
Image sat_image = Image::find_by_nickname( nicnam ) ;
if ( !sat_image ) {
    cout << "object not found" << endl;
    return 1 ;
}
int num_packets = sat_image.get_long("packetsReceived");
...
```

In this example, the Image that represents the managed object with the nickname NL-sat is returned by the call to `find_by_nickname`. The `get_long` function is called on this instance to get the value of the `packetsReceived` attribute of the managed object. For information on how to get the value of an attribute, refer to Section 5.6 “Getting Attribute Values From an Object” on page 5-17.

5.3.2.3 Getting and Setting Nicknames

Before you get or set the nickname of an Image instance, ensure that the Image instance is activated as explained in Section 5.2.2 “Activating the Instance of Image” on page 5-4.

To get the nickname of an Image instance, call the `get_nickname` function on the Image instance.

To set the nickname of an Image instance, call the `set_nickname` function on the Image instance.

5.4 Updating an Image Instance

Updating an `Image` instance loads attribute information from a managed object into the `Image` instance that represents the managed object. Each time you get or set attribute values of a managed object, or perform an action on a managed object, update the `Image` instance that represents it. Update the `Image` instance after you have initialized it.

To update an `Image` instance, call the `boot` function of the `Image` class. In the call to the `boot` function, specify a list of the attributes you want to get or set the values of.

Note – You must pass the list of attributes to the `boot` function in an array, even if you want to update only one attribute in the `Image` instance.

Code for loading attribute information into an `Image` instance is shown in CODE EXAMPLE 5-10.

CODE EXAMPLE 5-10 Updating an `Image` Instance

```
...
#include <pmi/hi.hh>                // High Level PMI
...
Array(DU) attrs;
attrs = Array(DU)(1);
attrs[0] = strdup("program");
if (!channel_image.boot(attrs)) {
    cout << channel_image.get_error_string() << endl;
    return FALSE;
}
Array(DU) dish_attrs;
dish_attrs = Array(DU)(1);
dish_attrs[0] = strdup("vchipId");
if (!dish_image.boot(dish_attrs)) {
    cout << dish_image.get_error_string() << endl;
    return FALSE;
}
...
```

In this example, arrays are defined as follows:

- An array named `attrs` contains the `program` attribute.
- An array named `dish_attrs` contains the `vchipId` attribute.

5.5 Deleting a Managed Object

To enable users of your applications to remove a managed resource from the network management environment, an application must delete the managed object that represents the managed resource.

An application can delete a managed object only if the specification of the managed object allows it to be deleted by a management operation. Specifically, the `NAME BINDING` clause of the GDMO specification of the managed object must contain the `DELETE` construct. Otherwise, any request to delete the managed object will be denied.

When an application attempts to delete a managed object that contains other managed objects, the result depends on how the `DELETE` construct is specified:

- If the `DELETES-CONTAINED-OBJECTS` modifier is applied to the `DELETE` construct, the managed object and all the managed objects it contains are deleted.
- Otherwise, the request to delete the managed object is denied. In that case, all the managed objects contained must be deleted before trying to delete the containing object.

Deleting a managed object involves:

- Selecting the managed object as described in Section 5.3 “Selecting a Managed Object” on page 5-8
- Removing the managed object from the MIS

5.5.1 Removing the Managed Object From the MIS

After you have specified the managed object you want to delete, remove it from the MIS by calling the `destroy` function of the `Image` class.

A call to the `destroy` function is shown in CODE EXAMPLE 5-11.

CODE EXAMPLE 5-11 Removing a Managed Object from the MIS

```
...
#include <pmi/hi.hh>           // High Level PMI
...
    if (!del_image.destroy())
        cout << "Deletion failed" << endl;
    else
        cout << "Deletion succeeded" << endl;
...
```

In this example, the overloaded NOT (!) operator acts on the call to `destroy` to verify that the managed object was removed from the MIS. If it was not removed, an error message explaining the reason for the failure is displayed.

5.5.2 Example Object Deletion Function

An example of a function that contains all the code for deleting a managed object is shown in CODE EXAMPLE 5-12.

CODE EXAMPLE 5-12 Example Object Deletion Function

```
...
#include <pmi/hi.hh>                // High Level PMI
#include <rw/cstring.h>             // Rogue Wave RWCString
...
Boolean delete_object(RWCString fdn)
{
    Image del_image(fdn.data());
    if (del_image.get_error_type() != PMI_SUCCESS) {
        cout << del_image.get_error_string() << endl;
        return FALSE;
    }

    // No need to waste time activating the Image object. If the
    // program reaches here, the Image object exists, so it can be
    // removed.
    if (!del_image.destroy())
        cout << "Deletion failed" << endl;
    else
        cout << "Deletion succeeded" << endl;
}
...
```

5.6 Getting Attribute Values From an Object

An application monitors managed resources by getting attribute values from managed objects that represent those managed resources.

An application can get an attribute only if the specification of the attribute allows it to be read by a management operation. Specifically, the property list in the `ATTRIBUTES` construct of the attribute's GDMO specification must include the `GET` operation.

Getting attribute values from a managed object involves:

- Selecting the managed object as described in Section 5.3 “Selecting a Managed Object” on page 5-8
- Loading attribute information into the `Image` instance as described in Section 5.4 “Updating an Image Instance” on page 5-14
- Getting attribute values from the `Image` instance

To get an attribute value from the `Image` instance, call one of the functions of the `Image` class listed in TABLE 5-1. The function to call depends on the data type of the attribute as defined in its ASN.1 module specification.

TABLE 5-1 Functions for Getting Attribute Values From an `Image` Instance

Data Type	Function
String	<code>get_str</code>
Integer	<code>get_long</code>
Real	<code>get_dbl</code>
Arbitrarily long integer	<code>get_gint</code>
Any complex ASN.1 type	<code>get_raw</code>

The `get_raw` function returns an instance of the `Morf` class. To process the `Morf` instance returned, call functions of the `Morf` class. For information on how to use the `Morf` class, refer to Chapter 9.

Each of the functions listed in TABLE 5-1 takes the name of attribute you want to get as a parameter. The attribute name must be specified exactly as it appears in the GDMO specification of the managed object. Attribute names are case sensitive.

If an attribute with the same name is defined in more than one of the GDMO documents loaded into the MIS, you must specify in which document the attribute you are interested in is defined.

To specify the document, prefix the attribute name with the document name specified in the `MODULE` construct of the managed object's GDMO specification, for example: `"My Document":reusedAttribute`.

Code for getting attribute values is shown in CODE EXAMPLE 5-13.

CODE EXAMPLE 5-13 Getting Attribute Values

```
...
#include <pmi/hi.hh>                // High Level PMI
#include <rw/cstring.h>             // Rogue Wave RWCString
...
int id_number = dish_image.get_long("vchipId");

Morf prog_info = channel_image.get_raw("program");
if (!prog_info.has_value()) {
    return TRUE;
}

RWCString rating;

// The program attribute is a choice between
// NULL or { program_name, rating }
// is_choice() will always be true if the Morf object is
// constructed properly
if (prog_info.is_choice()) {
    // Now extract the contents of the program
    Morf newm = prog_info.extract(DU());
    // If the program strings are in the attribute,
    // there will be a list of two elements
    if (newm.is_list()) {
        // Now get the required element
        Array(Morf) mm = newm.split_array();
        rating = mm[1].get_str().chp();
    } else
        return TRUE;
} else
    return TRUE;
...
```

In this example functions are called to get attributes as follows:

- The `get_long` function is called to get the integer attribute `vchipId`.
- The `get_raw` function is called to get the complex attribute `program`.

Functions of the `Morf` class are called to process the `Morf` object returned by the `get_raw` function. For information on how to use the `Morf` class, refer to Chapter 9.

5.7 Setting Attribute Values of an Object

To control managed resources, an application sets attribute values for managed objects that represent those managed resources.

An application can set an attribute only if the specification of the attribute allows it to be set by a management operation. Specifically, the property list in the `ATTRIBUTES` construct of the attribute's GDMO specification must include the operation you want to perform when setting the attribute. Otherwise, any request to set the attribute will be denied. For more information on attribute-setting operations, refer to Section 5.7.1.3 "Operation" on page 5-21.

Setting attribute values of an object involves:

- Selecting the managed object as described in Section 5.3 "Selecting a Managed Object" on page 5-8
- Loading attribute information into the `Image` instance as described in Section 5.4 "Updating an Image Instance" on page 5-14
- Setting attribute values in the `Image` instance
- Updating the MIS with the changed values

5.7.1 Setting Attribute Values in the Image Instance

After you have selected the managed object and loaded attribute information from the managed object into the `Image` instance, call functions of the `Image` class to set attributes in the `Image` instance. The function to call depends on the data type of the attribute as defined in its ASN.1 module specification. The functions for setting attribute values in an `Image` instance are listed in TABLE 5-2.

TABLE 5-2 Functions for Setting Attribute Values in an `Image` Instance

Data Type	Function
String	<code>set_str</code>
Integer	<code>set_long</code>
Real	<code>set_dbl</code>
Arbitrarily long integer	<code>set_gint</code>
Any complex ASN.1 type	<code>set_raw</code>

In the call to a function for setting an attribute value, you have to specify:

- The name of the attribute you want to set
- The value you want to set the attribute to
- The operation that you want to be carried out to set the attribute

Note – Calling a function for setting an attribute value changes only the value cached in your application. To change the value in the actual managed object, you must propagate the change to the managed object as explained in Section 5.7.2 “Updating the MIS With the Changed Values” on page 5-21.

5.7.1.1 Attribute Name

The attribute name must be specified exactly as it appears in the GDMO specification of the managed object. Attribute names are case sensitive.

If an attribute with the same name is defined in more than one of the GDMO documents loaded into the MIS, you must specify in which document the attribute you are interested in is defined. To specify the document, prefix the attribute name with the document name specified in the `MODULE` construct of the managed object's GDMO specification, for example: `"My Document":reusedAttribute`.

5.7.1.2 Attribute Value

The value you specify must be consistent with any restrictions specified in the property list in the `ATTRIBUTES` construct of the attribute's GDMO specification.

5.7.1.3 Operation

The operation specifies how the attribute value is to be modified. The operation you want to perform must be in the property list in the `ATTRIBUTES` construct of the attribute's GDMO specification. It must be one of the operations listed in TABLE 5-3.

TABLE 5-3 Operations for Setting Attributes

Operation	Result
REPLACE	Replaces the existing value with that specified in the function call. It corresponds to the <code>REPLACE</code> operation in a property list. <code>REPLACE</code> is the default operation.
INCLUDE	Adds the value specified in the function call to the current value of a multi-valued attribute. It corresponds to the <code>ADD</code> operation in a property list. Specify the <code>INCLUDE</code> operation for multi-valued attributes only. If you specify the <code>INCLUDE</code> operation for a single-valued attribute, an exception is thrown.
EXCLUDE	Removes the value specified in the function call from the current value of a multi-valued attribute. It corresponds to the <code>REMOVE</code> operation in a property list. Specify the <code>EXCLUDE</code> operation for multi-valued attributes only. If you specify the <code>EXCLUDE</code> operation for a single-valued attribute, an exception is thrown.
IGNORE	Specifies that the value specified in the function call is the initial value of the attribute. If the attribute has already been initialized, its value is not changed.
DEFAULT	Replaces the existing value with the default value defined in the property list in the <code>ATTRIBUTES</code> construct of the attribute's GDMO specification.

5.7.2 Updating the MIS With the Changed Values

After you have set attributes in the `Image` instance, update the MIS with the changed values to propagate the changes to the managed object itself. To update the MIS with the changed values, call the `store` function of the `Image` class.

5.7.3 Checking If the MIS Has Been Updated

Updating the MIS more frequently than necessary can impair the performance of your application. To enhance the performance of your application, check if the MIS has been updated before you update it with changed values.

To check if the MIS has been updated, get the value last set by your application and compare it to the value stored in the MIS. For information on how to get the value stored in the MIS, refer to Section 5.6 “Getting Attribute Values From an Object” on page 5-17.

To get the value last set by your application, call one of the functions of the `Image` class listed in TABLE 5-4. The function to call depends on the data type of the attribute as defined in its ASN.1 module specification.

TABLE 5-4 Functions for Getting the Value Last Set by an Application

Data Type	Function
String	<code>get_set_str</code>
Integer	<code>get_set_long</code>
Real	<code>get_set_dbl</code>
Arbitrarily long integer	<code>get_set_gint</code>
Any complex ASN.1 type	<code>get_set_raw</code>

The `get_set_raw` function returns an instance of the `Morf` class. To process the `Morf` instance returned, call functions of the `Morf` class. For information on how to use the `Morf` class, refer to Chapter 9.

Each of the functions listed in TABLE 5-4 takes the name of attribute you want to get as a parameter. The restrictions on these attribute names are identical to those given in Section 5.7.1.1 “Attribute Name” on page 5-20.

5.7.4 Real and Imaginary Values in an Image Instance

To enable you to compare the value last set by your application with the value stored in the MIS, an `Image` instance stores the following values for each attribute of a managed object:

- **A real value.** The real value represents the value in the MIS.
- **An imaginary value.** The imaginary value represents the value set by a call to one of the functions for setting attribute values listed in TABLE 5-2.

When the MIS is updated, the real value is set to the imaginary value. When an `Image` instance is activated, the real value is updated, but the imaginary value is left unchanged.

When you get an attribute value as described in Section 5.6 “Getting Attribute Values From an Object” on page 5-17, you retrieve the real value from the `Image` instance. When you get the value last set by your application as described in Section 5.7.3 “Checking If the MIS Has Been Updated” on page 5-21, you retrieve the imaginary value from the `Image` instance.

5.7.5 Example

Code for setting an attribute value is shown in CODE EXAMPLE 5-14.

CODE EXAMPLE 5-14 Setting an Attribute Value

```
...
#include <pmi/hi.hh>                // High Level PMI
...
// Prevent any further alarms generated in the system
// from being logged.
Image test_image("systemId="myhost"logId=string:\\"AlarmLog\\"");
    test_image.boot();
    test_image.set_str("administrativeState", "locked");
    test_image.store();
...
```

In this example the `administrativeState` attribute of the `AlarmLog` object is set to the text `locked`.

5.8 Performing an Action on an Object

An action is an operation that cannot be modelled by a pre-defined operation such as getting or setting an attribute. An action enables you to implement specialized behavior, for example, changing attributes of one or many objects in a single operation or providing the results of a query in a particular format.

Performing an action on an object involves:

- Selecting the managed object as described in Section 5.3 “Selecting a Managed Object” on page 5-8
- Loading attribute information into the `Image` instance as described in Section 5.4 “Updating an Image Instance” on page 5-14
- Sending the action request

To send an action request, call one of the functions of the `Image` class listed in TABLE 5-5, depending on the format of the action parameter.

TABLE 5-5 Functions for Sending an Action Request to a Managed Object

Format	Function	Comment
Text	<code>call</code>	The <code>call</code> function returns the action reply in text form.
Encoded	<code>call_raw</code>	You have to construct an instance of the <code>Morf</code> class to represent the action parameter. The <code>call_raw</code> function returns the action reply in encoded form as an instance of the <code>Morf</code> class.

In the call to `call` or `call_raw` you have to specify:

- **The name of the action you want to perform.** The action name must be specified exactly as it appears in the GDMO specification of the managed object. Action names are case sensitive.
- **The action parameter.** The syntax of the action parameter must be specified exactly as it appears in the GDMO specification of the managed object.

Code for sending an action request is shown in CODE EXAMPLE 5-15.

CODE EXAMPLE 5-15 Sending an Action Request

```

...
#include <pmi/hi.hh>                // High Level PMI
...
char * topo_name2Id(char * host, char * nodename)
{
    char dn[1024] = "";
    char action_name[100] = "topoNodeGetByName";
    char action_para[100];
    sprintf(action_para, "%s", nodename);
    sprintf(dn, "/systemId='%s'/topoNodeDBId=NULL", host);
    Image topo = Image(dn);

    ...
    Syntax syn_input = topo.get_param_syntax(action_name);
    ...
    Morf morf_input(syn_input, DU(action_para));
    ...
    Morf morf_result = topo.call_raw(DU(action_name), morf_input);
    if (morf_result.get_error_type() != PMI_SUCCESS) {
        cout << morf_result.get_error_string() << endl;
        exit(2);
    }

    ...
}

```

In this example, the `call_raw` function of the `Image` class is called in the body of a function named `topo_name2Id`. The name of the action to be performed is `topoNodeGetByName`.

The action parameter is a `Morf` object constructed as follows:

- The `get_param_syntax` function is called to obtain the syntax of the `topoNodeGetByName` action parameter.
- The following are passed to the constructor of the `Morf` class:
 - The `Syntax` object returned by `get_param_syntax`
 - The node name passed to the `topo_name2Id` function

If the call to `call_raw` is unsuccessful, the reason for the failure is printed.

For information on how to use the `Morf` class, see Chapter 9.

The GDMO specification of the `topoNodeGetByName` action is shown in CODE EXAMPLE 5-16.

CODE EXAMPLE 5-16 GDMO Specification of the `topoNodeGetByName` Action

```
...
topoNodeGetByName ACTION
    BEHAVIOUR topoNodeGetByNameBehaviour BEHAVIOUR DEFINED AS
        !This action returns the topoNodeId of the
        topoNode whose topoNodeName attribute matches the
        input name!;
    ;
    WITH INFORMATION SYNTAX EM-TOPO-ASN1.TopoNodeName;
    WITH REPLY SYNTAX EM-TOPO-ASN1.TopoNodes;
    REGISTERED AS { em-topo-action 1 };
...
```

The ASN.1 definitions of the data types used by the `topoNodeGetByName` action are shown in CODE EXAMPLE 5-17.

CODE EXAMPLE 5-17 ASN.1 Definitions of Data Types Used by `topoNodeGetByName`

```
...
TopoNodeName ::= GraphicString(SIZE(0..255))
TopoNodeId ::= INTEGER (0..4294967295)
TopoNodes ::= SET OF TopoNodeId
...
```

5.9 Tracking Changes to an Object

Tracking changes to an object updates an `Image` instance with changes to the managed object that the instance represents. Tracking changes ensures that your network management application has access to current data about your managed resources.

Depending on your requirements, you can track changes to an object automatically or manually. Automatically tracking changes requires you to write less code but gives you less control than manually tracking changes.

5.9.1 Automatically Tracking Changes to an Object

Automatically tracking changes to an object causes the MIS to update the `Image` instance whenever the MIS receives an attribute value change notification for that instance.

Track changes to an object automatically if you want to simplify the application development process (for example during the prototype phase) or if you do not require control over when an `Image` instance is updated.

To track changes to an object automatically, set the `TRACKMODE` property of the `Image` instance to `TRACK`. To set the `TRACKMODE` property of an `Image` instance, call the `set_prop` function of the `Image` class before activating the instance.

Code for setting the `TRACKMODE` property to `TRACK` is shown in CODE EXAMPLE 5-18.

CODE EXAMPLE 5-18 Setting the `TACKMODE` Property of an `Image` Instance

```
...
#include <pmi/hi.hh>                // High Level PMI
...
im.set_prop(duTRACKMODE, duTRACK);
...
```


5.9.2 Manually Tracking Changes to an Object

Track changes to an object manually if you need to control when an `Image` instance is updated. For example, if you need all attributes in an `Image` instance to have particular values before you perform an operation, manually track changes to the object that the `Image` instance represents.

Tracking changes to an object manually involves:

- Setting the `TRACKMODE` property of the `Image` instance to `SNAP`
- Using callback functions to update an `Image` instance with changes to the managed object that the instance represents

If you track changes to an object manually, you have to write code in your callbacks for updating the `Image` instance. Having to write your own code makes coding your application more complicated. But it gives you control over which changes you update the `Image` instance with, and enhances the performance of your application.

For information on how to use callback functions to update an `Image` instance, refer to Section 7.2 “Processing Information in Event Notifications” on page 7-4.

5.10 Retrieving Data From the Metadata Repository

The metadata repository (MDR) stores information about managed objects that is defined in your object model. Retrieve data from the metadata repository when:

- You want to verify what is already loaded into the MDR before loading an updated object model.
- You want your application to process attributes differently depending on their data types. For example, you want how an attribute is displayed to depend on its ASN.1 type.

The MDR is represented as a managed object in the Solstice EM MIS. To retrieve metadata from the MDR, an application must be able to perform an action on the MDR managed object.

Performing an action on the MDR managed object involves:

- Selecting the MDR managed object
- Updating the `Image` instance that represents the MDR managed object
- Sending the action request

5.10.1 Selecting the MDR Managed Object

When you perform an action on the MDR managed object, you have to select it by specifying its FDN. To select a managed object by specifying its FDN, create and initialize an instance of the `Image` class, specifying only the FDN. For more information, refer to Section 5.3.1 “Selecting a Managed Object by Specifying its FDN or LDN” on page 5-9.

The MDR managed object is contained by the `system` object and is named by the `metaName` attribute, which always has the value `MDR`. Therefore, the FDN of the MDR managed object is as follows:

```
/systemId= "host" /metaName= "MDR "
```

Where *host* is the host name of the MIS associated with the MDR.

Code for selecting the MDR managed object is shown in CODE EXAMPLE 5-19.

CODE EXAMPLE 5-19 Selecting the MDR Managed Object

```
...  
#include <hi.hh>                                // High Level PMI  
...  
    sprintf(dn, "/systemId=\"%s\"/metaName=\"%MDR\" ", server);  
    Image mdr = Image(dn);  
...
```

5.10.2 Updating the Image Instance That Represents the MDR Managed Object

After you have selected the MDR managed object, update the `Image` instance that represents it. To update an `Image` instance, call the `boot` function of the `Image` class. For more information, refer to Section 5.4 “Updating an Image Instance” on page 5-14.

Code for updating an `Image` instance that represents the MDR managed object is shown in CODE EXAMPLE 5-20.

CODE EXAMPLE 5-20 Updating the `Image` Instance that Represents the MDR

```
...
#include <hi.hh>                                // High Level PMI
...
// Activate the mdr object.
    if (!mdr.boot()) {
        cout << mdr.get_error_string() << endl;
        exit(2);
    }
...
```

5.10.3 Sending the Action Request

To send an action request to retrieve metadata from the MDR, call the `call` function of the `Image` class. For more information, refer to Section 5.8 “Performing an Action on an Object” on page 5-23. In the call to the `call` function, specify:

- **The name of the action you want to perform.** The action to use depends on the metadata you want to retrieve.
- **The action parameter.** The action parameter depends on the action you want to send.

The actions for retrieving metadata from the MDR are shown in TABLE 5-6.

TABLE 5-6 Actions for Retrieving Metadata From the MDR

Metadata	Action
The name of the ASN.1 module that an attribute is defined in	<code>getAttribute</code>
The name of each GDMO document loaded into the MDR	<code>getAllDocuments</code>
Complete information in text format about an ASN.1 module	<code>getAsn1Module</code>
Complete information in text format about a class	<code>getObjectClass</code>
A list of all items defined in a GDMO document and their OIDs	<code>getDocument</code>
The name of an item identified by an OID	<code>getOidName</code>

TABLE 5-6 Actions for Retrieving Metadata From the MDR (*Continued*)

Metadata	Action
A list of all packages that characterize a managed object class	getPackagesByOC
Information on the attributes, actions, and notifications defined in a package	getPackage
Information on the notifications and attributes, when OID for a particular notification is given.	getNotification AndAttributeIds

Code for sending an action request is shown in CODE EXAMPLE 5-21.

CODE EXAMPLE 5-21 Sending an Action Request

```
...  
#include <hi.hh>                               // High Level PMI  
#include <rw/cstring.h>                         // Rogue Wave RWCString  
...  
    // Send the action and get the result data.  
    DU mdr_data = mdr.call(action_name, action_para);  
...
```

In this example, the `call` function of the `Image` class is called to send an action request. The variable `action_name` specifies the name of the action to be performed. The variable `action_para` specifies the action parameter. The initialization of the `action_name` and `action_para` variables is not shown in the example.

5.11 Simulating an Agent Object

If the agent and manager applications are being developed simultaneously, you need to test your manager application separately from the agents it will manage. If you want to test your management application separately, you can use the MIS to simulate an agent.

Using the MIS to simulate an agent involves:

- Containing managed objects in the MIS
- Making read-only attributes modifiable
- Loading GDMO descriptions into the MIS
- Creating and modifying objects in the MIS

5.11.1 Containing Managed Objects in the Solstice EM MIS

In a live network management system, managed objects are contained in an agent. When the MIS is acting as an agent in a simulation, managed objects must be contained in the MIS itself. To enable managed objects to be contained in the MIS, managed object classes that would in a live system be instantiated directly under the agent object must be instantiated under the `system` object.

To allow a managed object class to be instantiated under the `system` object, define a name binding clause in which the superior object class is `system`. An example of such a name binding clause is given in CODE EXAMPLE 5-22.

CODE EXAMPLE 5-22 Name Binding Clause for Instantiation Under `system`

```
...
satellite-system NAME BINDING
    SUBORDINATE OBJECT CLASS satellite;
    NAMED BY
    SUPERIOR OBJECT CLASS "Rec. X.721 | ISO/IEC 10165-2 : 1992":system;
    WITH ATTRIBUTE satelliteId;
    BEHAVIOUR satellite-systemBehaviour BEHAVIOUR DEFINED AS
        ! For the test agent, create local instances of
        satellite under the system branch of the tree
        !;
    ;
    CREATE;
    DELETE ONLY-IF-NO-CONTAINED-OBJECTS;
    REGISTERED AS { satman-binding 1 };
...
```

In this example, the `satellite` managed object class can be instantiated under the `system` object.

5.11.2 Making Read-Only Attributes Modifiable

The GDMO specification of some attributes does not permit them to be modified by a management operation. For example, an attribute that represents the status of a device on the network is normally defined as a read-only attribute. However, if you want to use Solstice EM tools to modify such attributes when you are simulating an agent, you will need to make such attributes modifiable. To make a read-only attribute modifiable, add one of the following operations to the property list in the `ATTRIBUTES` construct of the attribute's GDMO specification:

- Replace
- Add
- Remove

5.11.3 Loading GDMO Descriptions Into the MIS

Load the GDMO descriptions of your managed object classes into the MIS to make them available to Solstice EM. For information on how to load the GDMO descriptions into the MIS, refer to Section 2.6 “Making Your Object Model Available to Solstice EM” on page 2-36.

5.11.4 Creating and Modifying Objects in the MIS

In a live network, managed objects are created and modified as a result of activity on the network. When you use the MIS to simulate an agent, use Solstice EM tools to simulate such activity by creating and modifying objects in the MIS. You can create and modify objects in the MIS interactively, or from the command line.

5.11.4.1 Creating and Modifying Objects Interactively

Creating and modifying objects interactively provides immediate verification of the attribute values you specify, thereby making it simple to set the values of complex ASN.1 types. To create and modify objects in the MIS interactively, use the MIS Objects tool.

5.11.4.2 Creating and Modifying Objects From the Command Line

Creating and modifying objects from the command line saves time and effort when you need to create and modify large numbers of objects, or when you need to repeat the same operations several times during testing.

To create and modify objects from the command line, use the `em_objop` utility. The `em_objop` utility creates and modifies objects in accordance with information supplied in an `em_objop` script, which is a text file.

Starting the `em_objop` Utility

To start the `em_objop` utility, type the following command:

```
prompt% em_objop -f file
```

where *file* is an `em_objop` script specifying how objects are to be created or modified.

Format of an `em_objop` Script

An `em_objop` script contains one or more commands. Each command specifies an operation to be carried out on an object. The format of a command is as follows:

```
operation
{
OPTION = 'encoding'
OC = moc
OI = moi
attr1 = val1
.
.
.
attrN = valN
}
```

The variable parts of this format are explained in TABLE 5-7.

TABLE 5-7 Variable Parts of the Format of an `em_objop` Script

<i>operation</i>	The operation you want to be performed. It must be one of the following keywords: <ul style="list-style-type: none">• CREATE - Create and initialize an object• SET - Set one or more attribute values of an object• DELETE - Delete an object• DERIVE - Derive an instance of the Album class
<i>encoding</i>	The encoding of the <code>em_objop</code> script. It must be one of the following keywords: <ul style="list-style-type: none">• HEX - Hexadecimal encoding

TABLE 5-7 Variable Parts of the Format of an `em_objop` Script (*Continued*)

	<ul style="list-style-type: none"> • OCTAL - Octal encoding
<i>moc</i>	The managed object class of the object that the operation is to be performed on.
<i>moi</i>	The fully distinguished name of the object that the operation is to be performed on. If <i>operation</i> is <code>DERIVE</code> , <i>moi</i> is a derivation string. For more information, refer to Section 6.3.2 “Format of a Derivation String” on page 6-5.
<i>attr1</i>	The name of the first attribute you want to set. Omit if <i>operation</i> is <code>DELETE</code> .
<i>val1</i>	The value that you want to set <i>attr1</i> to.
<i>attrN</i>	The name of the Nth attribute you want to set. Omit if <i>operation</i> is <code>DELETE</code> .
<i>valN</i>	The value that you want to set <i>attrN</i> to.

Example em_objop Scripts

Example `em_objop` scripts are given as follows:

- Creating an object - CODE EXAMPLE 5-23
- Setting an attribute value - CODE EXAMPLE 5-24
- Deleting an object - CODE EXAMPLE 5-25
- Deriving an Album instance - CODE EXAMPLE 5-26

CODE EXAMPLE 5-23 `em_objop` Script for Creating an Object

```
CREATE
{
OC=satellite
OI='satelliteId="NorthernLights" '
administrativeState=locked
operationalState=enabled
selfDestructCode='{{{ "T", 4, {{0 1}}}},{{ "R",8,{{1 3}}}},{{ "E",9,{{1 5}}}}}'
}
```

CODE EXAMPLE 5-24 `em_objop` Script for Setting an Attribute Value

```
SET
{
OC = 'autoManagementEntry'
OI = 'subsystemId="EM-MIS"/autoManagerId="TheAutoManager"/autoEntryId =
```


CODE EXAMPLE 5-24 em_objop Script for Setting an Attribute Value *(Continued)*

```
"MIBII_IsSnmpSystemUp_Host" '  
autoEntryTopoType = 'Ultra2'  
}
```

CODE EXAMPLE 5-25 em_objop Script for Deleting an Object

```
DELETE  
{  
OC = 'autoManagementEntry'  
OI = 'subsystemId="EM-MIS"/autoManagerId="TheAutoManager"/autoEntryId =  
"MIBII_IsSnmpSystemUp_Host" '  
}
```

CODE EXAMPLE 5-26 em_objop Script for Deriving an Album Instance

```
DERIVE  
{  
OC = 'autoManagementEntry'  
OI = 'subsystemId="EM-MIS"/autoManagerId="TheAutoManager"/LV(1) '  
}
```

5.12 Representing MIS Instances Locally in an Application

Using an instance of `Image` to store all the data in a managed object simplifies the coding of your applications, but it does require a lot of memory. If memory is scarce, you can save memory by defining your own C++ class to represent managed objects locally in an application.

Defining your own classes is particularly helpful when you have a managed object class that has many attributes and your application is controlling and monitoring only a small subset of them. In such a situation, you can create a class that has only the attributes that are relevant to your application.

Defining your own class makes coding your application more complicated. You have to write your own code for tracking changes, deletions, and creations. However, if your application is managing a large number of objects, you may need to define your own class to save memory.

If you are unsure if you need to define your own class, write a prototype application that uses the `Image` class, and test the performance of the prototype. Coding an application in this way is simple and rapidly provides useful performance data.

For example, the `satellite`, `channel`, and `dish` managed object classes in the `satellite` example have several attributes. But many of the example applications need only the identity, FDN, and position of a `satellite`, `channel`, or `dish` object. To save memory, the `Node` class is defined to hold important data common to the `satellite`, `channel`, and `dish` managed object classes.

For the `dish` managed object class, many of the example applications monitor only the `vchipId` attribute. To save memory, the `Dish` class is defined to hold this attribute.

CODE EXAMPLE 5-27 shows the definition of the `Node` and `Dish` C++ classes.

CODE EXAMPLE 5-27 C++ Representation of Managed Object Classes

```
class Node {
public:
    Node();
    Node(RWCString namestr, RWCString fdnstr);
    ~Node();

    int operator == (const Node &other) const {
        if (name == other.name)
            return(1);
        else
            return(0);
    }

    RWCString get_name();
    RWCString get_fdn();
    void get_location(Geo &lat, Geo &longit);

    void set_location(Geo lat, Geo longit);

    RWCString      name;
    RWCString      fdn;
    Coordinates    location;
};
```

CODE EXAMPLE 5-27 C++ Representation of Managed Object Classes (*Continued*)

```
class Dish : public Node {  
public:  
    Dish();  
    ~Dish();  
  
    Dish(Node newnode);  
  
    int vchip;  
  
};
```


Performing Management Operations on Object Collections

An object collection is a group of managed objects that your application can treat as a single entity. An object collection simplifies bulk operations by enabling you to select multiple managed objects to be the subject of a management operation. Any management operation that your application performs on an object collection is performed on every managed object in the object collection.

This chapter explains how to perform management operations on object collections.

- Section 6.1 “Grouping Managed Objects” on page 6-1
- Section 6.2 “Creating a Container for an Object Collection” on page 6-2
- Section 6.3 “Defining the Membership of an Object Collection” on page 6-3
- Section 6.4 “Tracking Changes to an Object Collection” on page 6-13
- Section 6.5 “Accessing All Objects in an Object Collection” on page 6-17
- Section 6.6 “Accessing Individual Objects in an Object Collection” on page 6-21
- Section 6.7 “Obtaining All Object Collections for an Object” on page 6-23

6.1 Grouping Managed Objects

You are free to choose the managed objects you group into an object collection. However, to ensure that an object collection you create is useful, group objects that are related in a way that is meaningful to your application.

Identify which managed objects are suitable for grouping into an object collection by:

- Considering the management operations your application will perform
- Analyzing your object model to find out which objects you want to perform the same management operation on simultaneously
- Identifying the management operations you want to perform on any object collections you create

For example, to lock all unlocked channels from a satellite, create an object collection that consists of all instances of the `channel` managed object class with the `administrativeState` attribute set to `unlocked` that are contained in a particular instance of `satellite`. After you have created the object collection, set the `administrativeState` attribute of all satellites in the object collection to `locked`.

Note – Solstice EM allows you to define a managed object to be a member of any number of object collections.

6.2 Creating a Container for an Object Collection

To group objects into an object collection, an application must create a container for the object collection. To create a container for an object collection, create and initialize an instance of the `Album` class. An instance of the `Album` class is a container for the managed objects in an object collection. It also provides functions for performing management operations on an object collection.

Solstice EM allows a managed object to be a member of any number of object collections.

Each managed object in an object collection is represented by an instance of the `Image` class. For more information on the `Image` class, refer to Chapter 5.

When you call the constructor of the `Album` class, you must specify a nickname for the `Album` instance. The nickname uniquely identifies the `Album` instance. It must be a text string. Spaces are permitted in a nickname.

Code for creating and initializing an instance of the `Album` class is shown in CODE EXAMPLE 6-1.

CODE EXAMPLE 6-1 Creating and Initializing an `Album` Instance

```
...  
#include <pmi/hi.hh>                                // High Level PMI  
...  
    satellites = Album("collection of all satellites");  
...
```

In this example, an `Album` instance is created and initialized with the nickname `collection of all satellites`.

6.3 Defining the Membership of an Object Collection

Defining the membership of an object collection selects managed objects to be grouped into the object collection. Depending where the managed objects are located in the management information tree (MIT), you can define the membership of an object collection by:

- **Derivation** to select managed objects in a subtree of the MIT
- **Enumeration** to select individual managed objects or object collections

If you want an object collection to contain a subtree of the MIT and individual objects or object collections, define the membership of the object collection by using a combination of derivation and enumeration.

6.3.1 Defining the Membership by Derivation

If the managed objects in your object collection are in a subtree of the MIT, define the membership of the object collection by derivation.

Defining the membership of an object collection by derivation involves:

- Setting a derivation string
- Starting the derivation

6.3.1.1 Setting a Derivation String

A derivation string selects one or more managed objects in a subtree of the MIT by specifying a base managed object, a scope, and a filter. To set a derivation string, call the `set_derivation` function of the `Album` class.

In the call to `set_derivation` you have to specify the derivation string. For details of the format of a derivation string, refer to Section 6.3.2 “Format of a Derivation String” on page 6-5.

Calling the `set_derivation` function does not cause your application to communicate with the MIS. Consequently, the `set_derivation` returns immediately after it is called, provided no errors occur.

Code for setting a derivation string is shown in CODE EXAMPLE 6-2.

CODE EXAMPLE 6-2 Setting a Derivation String

```
...
#include <pmi/hi.hh>                // High Level PMI
#include <rw/cstring.h>             // Rogue Wave RWCString
...
// Set up the distinguished name to start the derivation
RWCString derive_str;
derive_str = "/systemId=\"";
derive_str += server;
derive_str += "\"/LV(1)";
derive_str += "/CMISFilter(item:equality:{objectClass,satellite})";

cout << "Deriving satellites: " << derive_str.data() << endl;

if (!satellites.set_derivation((char *) derive_str.data())) {
    cout << "Failed to set derivation string." << endl;
    cout << satellites.get_error_string() << endl;
    return;
}
...
```

In this example, a derivation string is set up to select all instances of the `satellite` managed object class that are one level below the `system` object in the MIT. The `system` object is identified by the value of the `server` variable.

To enable the `server` variable for the host name to be included, the derivation string is constructed using a Rogue Wave `RWCString` object in this example. The `server` variable has already been initialized to a text string that contains the host name of the machine on which the MIS is running. The initialization of the `server` variable is not shown in this example.

6.3.1.2 Starting the Derivation

When you have set the derivation sting, start the derivation to define the membership of the object collection. To start the derivation, call the `derive` function of the `Album` class.

Calling the `derive` function causes your application to retrieve information from the MIS. If a large number of managed objects is selected by the derivation string, your application will become blocked for a long time while it waits for `derive` to return. If you want your application to continue with other processing during a lengthy derivation, start the derivation asynchronously as described in Chapter 8.

Code for starting a derivation is shown in CODE EXAMPLE 6-3.

CODE EXAMPLE 6-3 Starting a Derivation

```
...
#include <pmi/hi.hh>                // High Level PMI
...
    if (!satellites.derive()) {
        cout << "Derive failed." << endl;
        cout << satellites.get_error_string() << endl;
        return;
    }
...

```

6.3.2 Format of a Derivation String

A derivation string selects one or more managed objects in a subtree of the MIT by specifying a base managed object, a scope, and a filter. The format of a derivation string is as follows:

baseMO/scope/filter

Where:

- *baseMO* is the base managed object.
- *scope* is the scope.
- *filter* is the filter.

A forward slash(/) separates the base managed object from the scope, and the scope from the filter.

6.3.2.1 Base Managed Object

The base managed object is the root object of the subtree you want to select. Depending on the scope and the filter, the base managed object may not be one of the managed objects selected by the derivation string.

The base managed object is identified by one of the following:

- **Its fully distinguished name (FDN).** The first character in an FDN must be a forward slash.
- **Its local distinguished name (LDN).** The first character in an LDN may be any character *except* the forward slash.

The base managed object is optional. If you omit it, the `system` object is assumed. If you omit the base managed object, also omit the slash required to separate the base managed object from the scope.

6.3.2.2 Scope

The scope selects one or more managed objects in the subtree rooted at the base managed object. The scope is defined with reference to the base managed object. Set the scope in a derivation string to one of the values given in TABLE 6-1.

TABLE 6-1 Scope Values in a Derivation String

Value	Selects
ALL	The base managed object and its entire subtree.
LV(<i>n</i>)	Only level <i>n</i> subordinates of the base managed object, where <i>n</i> is an integer.
TO(<i>n</i>)	The base managed object and all its subordinates to level <i>n</i> , where <i>n</i> is an integer.
*	Only first-level subordinates of the base managed object. Equivalent to LV(1).
/	Only second-level subordinates of the base managed object. Equivalent to LV(2).
//*	Only third-level subordinates of the base managed object. Equivalent to LV(3).

The scope is optional. If you omit the scope, only the base managed object is selected.

If the subtree contains a large number of managed objects, setting the scope to `ALL` could cause your application to be timed out while it retrieves information from the MIS during a derivation. If possible, select fewer managed objects by changing the scope or by specifying a filter to prevent your application from being timed out.

The effects of these scope values are illustrated in FIGURE 6-1. In FIGURE 6-1, selected objects are shaded.

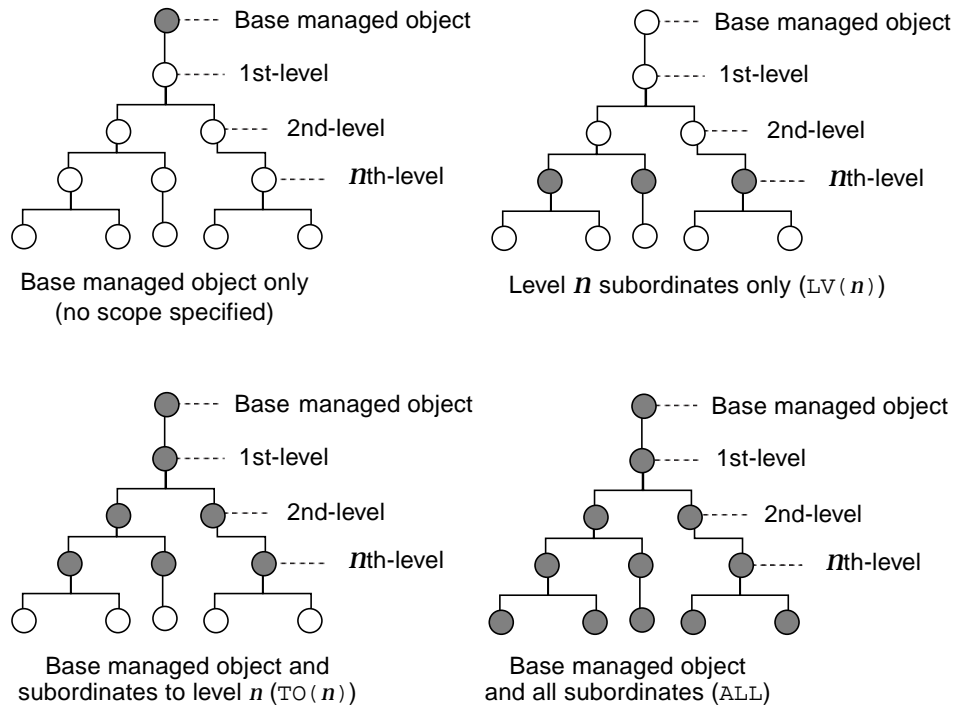


FIGURE 6-1 Scope Values

6.3.2.3 Filter

The filter selects or rejects objects based on the presence and values of specific attributes. The filter is a boolean expression, which may be a single test or a combination of multiple tests.

The filter is optional. If you omit it, all managed objects identified by the base managed object and scope are selected. If you omit the filter, also omit the slash required to separate the scope from the filter.

When a scope and a filter are combined, the scope is applied first, then the filter. An example of combining a scope and a filter is shown in FIGURE 6-2. In FIGURE 6-2, selected objects are shaded.

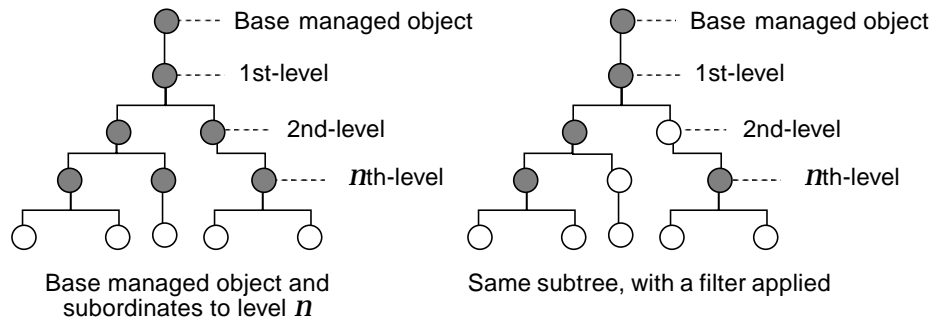


FIGURE 6-2 Combination of a Scope and a Filter

A filter contains an optional filter operator and one or more filter items. A filter item can in turn be a filter.

The format of a filter in a derivation string is as follows:

```
CMISFilter([ filterOperator: ] { filterItemList } )
```

Where:

- *filterOperator* is the filter operator. *filterOperator* is followed by a colon (:).
- *filterItemList* is a list of filter items. *filterItemList* is enclosed in braces. Each filter item in the list is separated by a comma.

The filter operator and the list of filter items are enclosed in parentheses.

Note – This format is the ASN.1 type `CMISFilter` that is defined in the ASN.1 module for ITU-T X.711/ISO-9596-1 *Common Management Information Protocol Specification* (`/opt/SUNWconn/em/etc/asn1/x711.asn1`).

Filter Operator

The filter operator is a logical operator for grouping elements in a filter that contains multiple filter items. Set the filter operator to one of the keywords given in TABLE 6-2.

TABLE 6-2 Filter Operator Keywords

Keyword	Meaning
and	Apply boolean <code>and</code> to the filter items that follow it.
or	Apply boolean <code>or</code> to the filter items that follow it.
not	Apply boolean <code>not</code> to the filter items that follow it. It can be applied only to one filter item.

The filter operator is optional. If you omit it, only one filter item is allowed in the filter. If you omit the filter operator, also omit the colon that follows it.

Filter Item

A filter item is a single test within a filter. The syntax of a filter item depends on whether you want to filter on substrings.

The syntax of a filter item without substrings is as follows:

```
item:comparison:{attributeId[,value]}
```

Where:

- *comparison* is the comparison made when the filter is applied. Set it to one of the keywords given in TABLE 6-3. *comparison* is followed by a colon (:).
- *attributeId* is the attribute identifier of the attribute to be tested when the filter is evaluated.
- *value* is the value against which the attribute specified in *attributeId* is tested when the filter is evaluated. It is not required when *comparison* is present. If you specify a value, it must be separated from *attributeId* by a comma (,).

The attribute identifier and the value are enclosed in braces.

TABLE 6-3 Comparison Keywords in a Filter Without Substrings

Keyword	Filter Item Evaluated to True if
equality	The managed object contains an attribute of type <i>attributeId</i> that has a value equal to <i>value</i> .
greaterOrEqual	<i>value</i> is greater than or equal to the value of an attribute of type <i>attributeId</i> that the managed object contains.
lessOrEqual	<i>value</i> is less than or equal to the value of an attribute of type <i>attributeId</i> that the managed object contains.
present	The managed object contains an attribute of type <i>attributeId</i> .
subsetOf	<i>value</i> is a subset of a set-valued attribute of type <i>attributeId</i> that the managed object contains.
supersetOf	<i>value</i> is a superset of a set-valued attribute of type <i>attributeId</i> that the managed object contains.
nonNullSetIntersection	The managed object contains a set-valued attribute of type <i>attributeId</i> and the intersection of its value and <i>value</i> is not empty.

The syntax of a filter item that includes substrings is as follows:

```
item:substrings:{substringItemList}
```

Where *substringItemList* is a list of substring items. *substringItemList* is enclosed in braces. Each substring item in the list is separated by a comma.

The syntax of a substring item is as follows:

```
part: {attributeId, value}
```

Where:

- *part* is the part of the attribute value that must match the value specified in the filter. Set it to one of the keywords given in TABLE 6-4. *part* is followed by a colon (:).
- *attributeId* is the attribute identifier of the attribute to be tested when the filter is evaluated.
- *value* is the value against which the attribute specified in *attributeId* is tested when the filter is evaluated.

The attribute identifier and the value are separated by a comma and enclosed in braces.

TABLE 6-4 Part Keywords in a Substring

Keyword	Filter-Item Evaluated to True if
initialString	The managed object contains an attribute of type <i>attributeId</i> the start of the value of which matches the string specified in <i>value</i> .
anyString	The managed object contains an attribute of type <i>attributeId</i> any part of the value of which matches the string specified in <i>value</i> .
finalString	The managed object contains an attribute of type <i>attributeId</i> the end of the value of which matches the string specified in <i>value</i> .

6.3.2.4 Example Derivation Strings

CODE EXAMPLE 6-4 shows the derivation string for selecting all log objects.

CODE EXAMPLE 6-4 Selecting All log Objects

```
LV(1)/CMISFilter(item: equality: {objectClass, log})
```

The object model defines that instances of log are contained in the system object. Therefore, the base managed object is not specified in this example because it is the system object, which is the default.

CODE EXAMPLE 6-5 shows the derivation string for selecting all enabled log objects.

CODE EXAMPLE 6-5 Selecting All Enabled Instances of log

```
LV(1)/CMISFilter(  
  and: {  
    item: equality: {objectClass, log},  
    item: equality: {operationalState, enabled}  
  }  
)
```

In this example, two filter items are combined using the and filter operator. The first filter item selects all log objects. The second filter item tests the value of the operationalState attribute to select only log objects that are enabled.

CODE EXAMPLE 6-6 shows the derivation string for selecting all objects that are not instances of the `log` class.

CODE EXAMPLE 6-6 Selecting all Objects That are not Instances of `log`

```
ALL/CMISFilter(not:{item:equality:{objectClass,log}})
```

The derivation strings shown in CODE EXAMPLE 6-7 are all equivalent.

CODE EXAMPLE 6-7 Equivalent Derivation Strings

```
/systemId="mako"/LV(1)/CMISFilter(item:equality:{objectClass,log})  
  
LV(1)/CMISFilter(item:equality:{objectClass,log})  
  
*/CMISFilter(item:equality:{objectClass,log})
```

In this example, each of the derivation strings selects all `log` objects that are one level below the `system` object in the MIT.

6.3.3 Defining the Membership by Enumeration

If the managed objects in your object collection are distributed throughout the MIT, or if you want to add an existing object collection, define the membership of the object collection by enumeration.

Note – If you define the membership of an object collection by enumeration, you have to maintain the membership of the object collection manually. For more information, refer to Section 6.4.1 “Maintaining the Membership of an Object Collection” on page 6-13.

To define the membership of the object collection by enumeration, call the `include` function of the `Album` class. The `include` function adds a managed object or an object collection to an object collection.

In the call to the `include` function you have to specify one of the following:

- The `Image` instance that represents the managed object you want to add
- The `Album` instance that contains the object collection you want to add

6.4 Tracking Changes to an Object Collection

Tracking changes to an object collection updates an `Album` instance with changes to the object collection that the instance contains. Tracking changes ensures that:

- Your network management application has access to current data about your managed resources.
- Any network management operations you perform on an object collection are performed on the required group of managed objects.

Tracking changes to an object collection involves:

- Maintaining the membership of an object collection
- Setting the mode of an object collection

6.4.1 Maintaining the Membership of an Object Collection

Maintaining the membership of an object collection ensures that the object collection accurately reflects the current state of your network. In a live network, managed objects are continually created and deleted as managed resources are added to and removed from the network management environment. Attribute values change as a result of activity on the network. All of these changes affect the membership of an object collection.

Depending on your performance requirements, you can maintain the membership of an object collection automatically or manually. Automatically maintaining the membership requires you to write less code but is less efficient than manually tracking changes. Furthermore, when you maintain the membership of an object collection automatically, all historical information on the membership the object collection is lost.

6.4.1.1 Automatically Maintaining the Membership of an Object Collection

Automatically maintaining the membership of an object collection causes the `Album` instance that contains the object collection to update the object collection by:

- Adding an object that satisfies the selection criteria of the object collection whenever such an object is created
- Removing an object that has been deleted
- Adding or removing an object if changes to its attribute values make it eligible or ineligible to be a member of the object collection

Maintain the membership of an object collection automatically in any of the following circumstances:

- You want to simplify the application development process (for example during the prototype phase).
- The performance of your application is not critical.
- You do not require historical information on the membership the object collection.

Note – If you want to maintain the membership of the object collection automatically, you must define the membership of the object collection by derivation. For more information, refer to Section 6.3.1 “Defining the Membership by Derivation” on page 6-3.

To maintain the membership of an object collection automatically, set the `TRACKMODE` property of the `Album` instance to `TRACK`. To set the `TRACKMODE` property of an `Album` instance, call the `set_prop` function of the `Album` class before defining the membership of the object collection that the instance contains.

Code for setting the `TRACKMODE` property to `TRACK` is shown in CODE EXAMPLE 6-8.

CODE EXAMPLE 6-8 Setting the `TACKMODE` Property of an `Album` Instance

```
...
#include <pmi/hi.hh>                                // High Level PMI
...
    // Automatically add Image objects to the Album object
    // based on derivation rules
    //
    if (!satellites.set_prop(duTRACKMODE, duTRACK)) {
        cout << satellites.get_error_string() << endl;
        return;
    }
...
```

6.4.1.2 Manually Maintaining the Membership of an Object Collection

Maintain the membership of an object collection manually if you need fast response from your application, or if you need to preserve historical information on the membership the object collection.

Maintaining the membership of an object collection manually involves:

- Setting the `TRACKMODE` property of the `Album` instance to `SNAP`
- Using callback functions to update the membership of the object collection

If you maintain the membership of an object collection manually, you have to write in your callback functions code for adding objects to or removing objects from the object collection. Having to write your own code makes coding your application more complicated. But it gives you control over which changes you update the object collection with, and enhances the performance of your application.

Use the following functions of the `Album` class in your callbacks to add objects to or remove objects from an object collection:

- `include` - To add an object or an object collection to an object collection
- `exclude` - To remove an object or an object collection from an object collection

In the call to the `include` or `exclude` function you have to specify one of the following:

- The `Image` instance that represents the object you want to add or remove
- The `Album` instance that contains the object collection you want to add or remove

Note – Every time you call the `derive` function you lose all information on the membership of the previous version of the object collection. To avoid losing any historical information, maintain the membership of the object collection by calling only the `include` and `exclude` functions.

Code for using callback functions to update the membership of the object collection is shown in CODE EXAMPLE 6-9.

CODE EXAMPLE 6-9 Using Callback Functions With an Object Collection

```
...
#include <pmi/hi.hh>                // High Level PMI
#include <rw/cstring.h>            // Rogue Wave RWCString
...
    // Register for callbacks when a new Image object is deleted.
    //
    if (!satellites.when("OBJECT_DESTROYED", Callback(remove_cb,0))) {
        cout << satellites.get_error_string() << endl;
        return;
    }
...
void remove_cb( Ptr, Ptr calldata)
{
    CurrentEvent ce(calldata);
    ce.do_something();
    cout << "*****Removing from the Album ";
    RWCString event_class(ce.get_objclass().chp());
    Image tmpimage(ce.get_objname());
    if (event_class.contains("dish")) {
```

```

        cout << "dishes*****" << endl;
        dishes.exclude(tmpimage);
    } else if (event_class.contains("channel")) {
        cout << "channels*****" << endl;
        channels.exclude(tmpimage);
    }
    ...
}

```

In this example, the `remove_cb` callback is called whenever the application receives a notification that a managed object has been deleted. When the `remove_cb` callback is called, the application extracts from the event information required to determine if the managed object is a member of an object collection maintained by the application. If it is, the `exclude` function of the `Album` class is called to remove the object from the object collection.

For more information on how to use callback functions, refer to Section 7.2 “Processing Information in Event Notifications” on page 7-4.

6.4.2 Setting the Mode of an Object Collection

The mode of an object collection specifies how `Image` instances are activated and tracked when they are added to an object collection. The mode is one of the following:

- **Automatic.** When the mode is set to automatic:
 - The `Image` instance that represents the managed object is activated when the object is added to the object collection.
 - Changes to the managed object are tracked automatically, even if the properties of `Image` instance specify manual tracking.
- **Manual.** When the mode is set to manual:
 - You must explicitly activate the `Image` instance that represents the object.
 - You must use the properties of the `Image` instance to specify whether changes to the object are tracked automatically or manually.

For more information, refer to Section 5.2.2 “Activating the Instance of `Image`” on page 5-4 and Section 5.9 “Tracking Changes to an Object” on page 5-26.

To set the mode of an object collection, set the `AUTOIMAGE` property of the `Album` instance as follows:

- YES to set the mode to automatic
- NO to set the mode to manual

To set the `AUTOIMAGE` property of an `Album` instance, call the `set_prop` function of the `Album` class before defining the membership of the object collection that the instance contains.

Code for setting the mode of an object collection is shown in CODE EXAMPLE 6-10.

CODE EXAMPLE 6-10 Setting the Mode of an Object Collection

```
...
#include <pmi/hi.hh>                                // High Level PMI
...
    // By default, an Image object is in the down state and
    // read only.
    // Specify automatic activation of new Image objects
    // which are children of the derivation FDN
    //
    if (!satellites.set_prop(duAUTOIMAGE, duYES)) {
        cout << satellites.get_error_string() << endl;
        return;
    }
...
```

In this example, the overloaded NOT (!) operator acts on the call to `set_prop` to verify that the mode of the object collection was set to automatic. If the mode was not set, an error message explaining the reason for the failure is displayed.

6.5 Accessing All Objects in an Object Collection

Accessing all objects in an object collection enables you to perform the same management operation on several managed objects simultaneously. You access all the objects in an object collection by performing management operations on the object collection. Any management operation performed on an object collection is performed on every managed object in the object collection.

The following management operations are permitted on an object collection:

- Adding all objects in an object collection to the management information server (MIS)
- Deleting all objects in an object collection
- Setting attribute values of all objects in an object collection
- Performing an action on all objects in an object collection

These management operations are described in detail for individual managed objects in Chapter 5.

When you perform a management operation on an object collection you have to set the synchronization of the object collection. The synchronization specifies how the collection reacts to a management operation that only some objects in the collection are able to perform.

6.5.1 Adding All Objects in an Object Collection to the MIS

Each object in an object collection is represented by an instance of the `Image` class. An instance of the `Image` class is a local representation of an object. A local representation is cached within your network management application. The actual object is in the MIS. If the managed objects in a collection do not already exist in the MIS, you can add them to the MIS in a single operation. To add all objects in an object collection to the MIS, call the `all_create` function of the `Album` class.

In the call to the `all_create` function, you can set an optional timeout to specify the maximum length of time allowed for adding an object in the object collection to the MIS. The timer used for this timeout is reset each time an object in the object collection is added to the MIS.

6.5.2 Deleting All Objects in an Object Collection

Deleting all objects in an object collection removes the managed resources represented by the objects in the collection from the network management environment. The conditions under which an object in an object collection is permitted to be deleted are defined in the GDMO specification of the managed object. For more information, refer to Section 5.5 “Deleting a Managed Object” on page 5-15.

To delete all objects in an object collection, call the `all_destroy` function of the `Album` class. The `all_destroy` function removes all objects in the object collection from the MIS.

In the call to the `all_destroy` function, you can set an optional timeout to specify the maximum length of time allowed for deleting an object in the object collection. The timer used for this timeout is reset each time an object in the object collection is deleted.

Note – The `Album` instance that contains the object collection remains in existence after all objects in the object collection have been deleted.

6.5.3 Setting Attribute Values of All Objects in an Object Collection

To control the managed resources represented by the objects in an object collection, an application sets attribute values of all objects in the collection. Setting attribute values of all objects in an object collection applies the same control operation to all objects in the collection.

The conditions under which an object attribute is permitted to be set are defined in the GDMO specification of the managed object. For more information, refer to Section 5.7 “Setting Attribute Values of an Object” on page 5-19.

Setting an attribute value of all objects in an object collection involves:

- Setting an attribute value in the `Image` instances
- Updating the MIS with the changed values

6.5.3.1 Setting an Attribute Value in the `Image` Instances

To set an attribute value in the `Image` instances that represent the objects in an object collection, call one of the functions of the `Album` class listed in TABLE 6-5, depending on the data type of the attribute.

TABLE 6-5 Functions for Setting Attribute Values in an Object Collection

Data Type	Function
String	<code>all_set_str</code>
Integer	<code>all_set_long</code>
Real	<code>all_set_dbl</code>
Arbitrarily long integer	<code>all_set_gint</code>
Any complex ASN.1 type	<code>all_set_raw</code>

In the call to a function for setting an attribute value, you have to specify:

- The name of the attribute you want to set (see Section 5.7.1.1 “Attribute Name” on page 5-20)
- The value you want to set the attribute to (see Section 5.7.1.2 “Attribute Value” on page 5-20)
- The operation that you want to be carried out to set the attribute (see Section 5.7.1.3 “Operation” on page 5-21)

6.5.3.2 Updating the MIS With the Changed Values

After you have set the attribute in the `Image` instances, update the MIS with the changed values to propagate the changes to the managed objects themselves. To update the MIS with the changed values, call the `all_store` function of the `Album` class.

6.5.4 Performing an Action on an All Objects in an Object Collection

An action is an operation that cannot be modelled by a pre-defined operation such as getting or setting an attribute. An action enables you to implement specialized behavior, for example, changing attributes of one or many objects in a single operation or providing the results of a query in a particular format.

To send an action request to an object collection, call the `all_call` function of the `Album` class.

In the call to `all_call`, specify:

- **The name of the action you want to perform.** The action name must be specified exactly as it appears in the GDMO specification of the managed object. Action names are case sensitive.
- **The action parameter.** The syntax of the action parameter must be as specified exactly in the GDMO specification of the managed object.
- **An optional timeout.** The timeout specifies the maximum length of time allowed for performing the action on an object in the object collection. The timer used for this timeout is reset each time the action is performed on an object in the object collection.

6.5.5 Setting the Synchronization of an Object Collection

The synchronization of an object collection specifies whether a management operation has to be successful for all the objects in the object collection if it is to be performed. The synchronization is one of the following:

- **Best effort.** The operation is performed even if it is successful only for some of the objects in the object collection.
- **Atomic.** The operation is performed only if all objects in the object collection can perform it.

To set the synchronization of an object collection, set the `BEST_EFFORT` property of the `Album` instance as follows:

- `YES` to set the synchronization to best effort
- `NO` to set the synchronization to atomic

To set the `BEST_EFFORT` property of an `Album` instance, call the `set_prop` function of the `Album` class before defining the membership of the object collection that the instance contains.

6.6 Accessing Individual Objects in an Object Collection

Access individual objects in an object collection if you want to:

- Perform a management operation on some, but not all, objects in an object collection
- Perform a management operation that must be performed separately on each object in an object collection (for example incrementing the value of the same attribute of all objects in an object collection)

Accessing individual objects in an object collection also provides information on the membership of an object collection.

To access individual objects in an object collection, use the `AlbumImage` class to retrieve each object from an object collection. An instance of the `AlbumImage` class represents the current `Image` instance in a list of `Image` instances or the current `Album` instance in a list of `Album` instances.

Retrieving each object from an object collection involves:

- Creating and initializing an instance of the AlbumImage class
- Calling the `first_image` function of the Album class to retrieve the first object in the object collection
- Calling the `next_image` function of the AlbumImage class to retrieve the second and later objects in the object collection

Code for retrieving each object from an object collection is shown in
CODE EXAMPLE 6-11.

CODE EXAMPLE 6-11 Retrieving Objects From an Object Collection

```
...
#include <pmi/hi.hh>                                // High Level PMI
...
Album          dishes;                             // Collection of dish objects
...
// Get and print each object in the Album object.
AlbumImage ali;
for (ali = dishes.first_image(); ali; ali = ali.next_image()) {
    Image im(ali);
    if (im.get_error_type() != PMI_SUCCESS) {
        cout << im.get_error_string() << endl;
        exit(8);
    }
    DU objname = im.get_objname();

    cout << endl;
    cout << "Dish Name: ";
    cout << objname.chp() << endl;
}
...
```

In this example, an instance of Image is created for each object in the object collection as follows:

- A `for` loop is initialized with the first object in an object collection.
- The `next_image` function of the AlbumImage class is called in the `for` loop to retrieve the second and later objects in the object collection.
- Each of the AlbumImage instances returned by the `first_image` and `next_image` functions is passed in turn to the copy constructor of Image.

The FDN of each Image instance created is obtained by calling the `get_objname` function of the Image class. Each FDN obtained is printed.

When the final Image instance has been reached, the `next_image` function returns NULL, thereby terminating the `for` loop.

6.7 Obtaining All Object Collections for an Object

Obtaining all object collections for an object keeps track of which object collections a managed object is a member of. By keeping track of which object collections a managed object is a member of, you can exclude a managed object from a collection when you no longer want operations on the collection to be applied to the managed object.

Obtaining all object collections for an object involves:

- Creating and initializing an instance of the `AlbumImage` class
- Calling the `first_album` function of the `Image` class to retrieve the first object collection that the object is a member of
- Calling the `next_album` function of the `AlbumImage` class to retrieve the second and later object collections that the object is a member of

Code for obtaining all object collections for an object is shown in CODE EXAMPLE 6-12.

CODE EXAMPLE 6-12 Obtaining all Object Collections for an Object

```
...
#include <pmi/hi.hh>                                // High Level PMI
...

Image img (DU(fdn)); // Managed object already exists, so no MOC
...
// Get and print each Album instance that contains the object
AlbumImage albimg;
for (albimg = img.first_album(); albimg; albimg = albimg.next_album()) {
    Album alb(albimg);
    if (alb.get_error_type() != PMI_SUCCESS) {
        cout << alb.get_error_string() << endl;
        exit(8);
    }

    DU alname = alb.get_prop(duNICKNAME);

    cout << endl;
    cout << "Album Nickname: ";
    cout << alname.chp() << endl;
}
...
```

In this example, an instance of `Album` is created for each object collection that the object is a member of as follows:

- A `for` loop is initialized with the first object collection that the object is a member of.
- The `next_image` function of the `AlbumImage` class is called in the `for` loop to retrieve the second and later object collections that the object is a member of.
- Each of the `AlbumImage` instances returned by the `first_album` and `next_album` functions is passed in turn to the copy constructor of `Album`.

The nickname of each `Album` instance created is obtained by calling the `get_prop` function of the `Album` class. Each nickname obtained is printed.

When the final `Album` instance has been reached, the `next_album` function returns `NULL`, thereby terminating the `for` loop.

Handling Events

Any network management application that monitors and controls managed resources on a network needs to process information it receives from those managed resources. Such information is contained in event notifications. An event notification is an unsolicited message sent from a managed object that represents a managed resource. Event notifications contain error information and other types of status information.

This chapter explains how to enable applications to handle event notifications.

- Section 7.1 “Event Notifications” on page 7-1
- Section 7.2 “Processing Information in Event Notifications” on page 7-4
- Section 7.3 “Scheduling Event Handling” on page 7-11
- Section 7.4 “Filtering Events” on page 7-16
- Section 7.5 “Simulating an Event” on page 7-20
- Section 7.6 “Subscribing to Log Record Events” on page 7-23

7.1 Event Notifications

An event notification is an unsolicited message sent from a managed object which represents a managed resource. The meaning of an event and the information contained in it depend on the event type. The event types supported by a managed object are defined in the GDMO specification of the managed object class. The event types themselves are defined in the object model of the managed system.

The definition of the events supported by the `satellite` managed object class in the sample programs is shown in CODE EXAMPLE 7-1.

CODE EXAMPLE 7-1 Specification of Event Types Supported by a Managed Object Class

```
satellite MANAGED OBJECT CLASS
...
CHARACTERIZED BY satellitePackage;
...
satellitePackage PACKAGE
...
NOTIFICATIONS
    "Rec. X. 721 | ISO/IEC 10165-2 : 1992":objectCreation,
    "Rec. X. 721 | ISO/IEC 10165-2 : 1992":objectDeletion,
    "Rec. X. 721 | ISO/IEC 10165-2 : 1992":attributeValueChange;
;
```

The GDMO specification of the objectCreation event is shown in CODE EXAMPLE 7-2.

CODE EXAMPLE 7-2 GDMO Specification of the objectCreation Event

```
objectCreation    NOTIFICATION
    BEHAVIOUR      objectCreationBehaviour;
    WITH INFORMATION SYNTAX Notification-ASN1Module.ObjectInfo
    AND ATTRIBUTE IDS
        sourceIndicator      sourceIndicator,
        attributeList         attributeList,
        notificationIdentifier notificationIdentifier,
        correlatedNotifications correlatedNotifications,
        additionalText         additionalText,
        additionalInformation additionalInformation;

REGISTERED AS      {joint-iso-ccitt ms(9) smi(3) part2(2) notification(10) 6};
-- changed by Technical Corrigendum 2

objectCreationBehaviour
BEHAVIOUR
    DEFINED AS "This notification type is used to report the creation of a
        managed object to another open system.";
```

Standards bodies such as the International Telecommunication Union - Telecommunication Standardization Sector (ITU-T) have written GDMO specifications for events that commonly occur in network management. Before defining your own event types, consult published network management standards to see if the event type you require has already been defined.

For example, recommendation ITU-T X.721/ISO-10165-2 *Definition of Management Information* defines the event types given in TABLE 7-1.

TABLE 7-1 Event Types Defined in Recommendation ITU-T X.721/ISO-10165-2

Event Type	Meaning
attributeValueChange	Reports changes to an attribute such as: <ul style="list-style-type: none"> • Addition or deletion of members to one or more multivalued attributes • Replacement of the value of one or more attributes • Setting attribute values to their defaults
communicationsAlarm	Reports when an object detects a communications error.
environmentalAlarm	Reports a problem in the environment.
equipmentAlarm	Reports a failure in the equipment.
integrityViolation	Reports that a potential interruption in information flow has occurred such that information may have been illegally modified, inserted, or deleted.
objectCreation	Reports the creation of a managed object to another open system.
objectDeletion	Reports the deletion of a managed object to another open system.
operationalViolation	Reports that the provision of the requested service was not possible due to the unavailability, malfunction, or incorrect invocation of the service.
physicalViolation	Reports that a physical resource has been violated in a way that indicates a potential security attack.
processingErrorAlarm	Reports processing failure in a managed object.
qualityofServiceAlarm	Reports a failure in the quality of service of the managed object.
relationshipChange	Reports a change in the value of one or more relationship attributes of a managed object. The change results from either internal operation of the managed object or via management operation.
securityServiceOrMechanismViolation	Reports that a security attack has been detected by a security service or mechanism.
stateChange	Reports a change in the value of one or more state attributes of a managed object. The change results from either internal operation of the managed object or via management operation.
timeDomainViolation	Reports that an event has occurred at an unexpected or prohibited time.

7.2 Processing Information in Event Notifications

To monitor and control managed resources, a network management application needs to process information it receives in event notifications. Processing the information in event notifications involves:

- Registering callback functions
- Writing callback functions
- Controlling event-related updates

7.2.1 Registering Callback Functions for Event Handling

Registering a callback function associates the callback function with a particular event type. Register a callback function for each event type you want to process. To register a callback function, call the `when` function of one of the following classes:

- `Platform` to handle events from all object instances in the management information server (MIS)
- `Image` to handle events from a single object instance
- `Album` to handle events from an object collection

In the call to the `when` function you have to:

- Specify the event type
- Initialize an instance of the `Callback` class to represent the callback function associated with the event type

7.2.1.1 Specifying the Event Type

If the event type is recognized by the `when` function, specify the shorthand form given in TABLE 7-2. If the event type is not recognized by the `when` function, specify the fully qualified name defined in the GDMO document.

TABLE 7-2 Event Types Recognized by the `when` Function

Event Type	Shorthand Form	Class	Meaning
All	RAW_EVENT	Album, Image, Platform	An event has occurred. The callback is called when an event of any type is received.
"Rec. X.721 ISO/IEC 10165-2:1992": attributeValueChange	ATTR_CHANGED	Image, Platform	An attribute value of an object has changed.
"Rec. X.721 ISO/IEC 10165-2:1992": objectCreation	OBJECT_CREATED	Album, Image, Platform	An object has been created.
"Rec. X.721 ISO/IEC 10165-2:1992": objectDeletion	OBJECT_DESTROYED	Album, Image, Platform	An object has been destroyed.
Not defined in a GDMO specification	DISCONNECTED	Platform	The MIS has become disconnected from the application.
Not defined in a GDMO specification	IMAGE_INCLUDED	Album	An Image instance was added to this Album instance.
Not defined in a GDMO specification	IMAGE_EXCLUDED	Album	An Image instance was deleted from the Album instance. This is an internal event, and has no MIS event information associated with it.
Not defined in a GDMO specification	WAIT	Platform	The PMI is entering a wait state. The callback is called again when leaving the wait state, but with a null name.

7.2.1.2 Initializing an Instance of the Callback Class

To initialize an instance of the `Callback` class, call its constructor in the call to the `when` function. In the call to the constructor of the `Callback` class, you must specify:

- The name of the callback function.
- A pointer to the data to be passed as an argument to the callback function. You can specify a null pointer if you do not want to pass any data to the callback function.

7.2.1.3 Example

Code for registering callback functions is shown in CODE EXAMPLE 7-3.

CODE EXAMPLE 7-3 Registering Callback Functions

```
...
#include <pmi/hi.hh>                                //High Level PMI
...
plat.when("RAW_EVENT", Callback(raw_cb,0)); //Register event
plat.when("DISCONNECTED",Callback(disc_cb,0)); //callback routines
...
```

In this example, the callback function `raw_cb` is registered so that it is called when the application receives an event of type `RAW_EVENT`. The callback function `disc_cb` is registered so that it is called when the application receives an event of type `DISCONNECTED`. The `raw_cb` callback function is also called, because an event of type `DISCONNECTED` is also an event of type `RAW_EVENT`. No data are passed to either callback function.

7.2.2 Writing Callback Functions for Event Handling

Write a callback function to handle each type of event you want to process. Each callback function you write must contain the code needed to carry out the processing you require. Writing a callback function involves:

- Defining the signature of the callback function
- Extracting information from an event notification

7.2.2.1 Defining the Signature of the Callback Function

The signature of any callback function you write must be in the following format:

```
void callbackName (Ptr userdata, Ptr calldata)
```

Where:

- *callbackName* is the name you assign to your callback function.
- *userdata* is the data passed to the callback by your application.
- *calldata* is the data passed to the callback by the scheduler that calls the callback function.

7.2.2.2 Extracting Information From an Event Notification

Extract the information from event notifications that is useful to users of your application. To extract information from an event notification, call functions of the `CurrentEvent` class in the body of your callback function. An application represents every event notification it receives as an instance of the `CurrentEvent` class.

The functions for extracting information from an event are listed in TABLE 7-3. They extract the following types of information:

- **Event information**, which is information on what caused the event notification to be generated
- **Contextual information**, which is information on the event notification itself

TABLE 7-3 Functions for Extracting Information from Event Notifications

Information	Function
Event information	
The entire event as text.	<code>get_event</code>
The entire event in encoded form.	<code>get_event_raw</code>
The event information as text.	<code>get_info</code>
The event information in encoded form.	<code>get_info_raw</code>
The name of the event. This name is the human-readable form of the event object identifier (OID). It indicates the event type as defined in the GDMO specification of the event, for example: "Rec. X.721 ISO/IEC 10165-2 :1992":objectDeletion	<code>get_name</code>
The event OID defined in the GDMO specification of the event.	<code>get_oid</code>
A pointer to the event. This pointer has no meaning outside the scope of the <code>CurrentEvent</code> object that represents the event notification.	<code>get_message</code>
Contextual information	
The <code>Album</code> instance associated with the event.	<code>get_album</code>
The event type as specified in the call to the <code>when</code> function that registers the callback for the event.	<code>get_eventtype</code>
The <code>Image</code> instance associated with the event.	<code>get_image</code>
The class of the managed object represented by the <code>Image</code> instance associated with the event.	<code>get_objclass</code>
The fully distinguished name (FDN) of the managed object represented by the <code>Image</code> instance associated with the event.	<code>get_objname</code>
The location of the MIS associated with the event.	<code>get_platform</code>
The time of the event.	<code>get_time</code>

A callback function is shown in CODE EXAMPLE 7-4.

CODE EXAMPLE 7-4 Callback Function

```
...
#include <pmi/hi.hh>                // High Level PMI
#include <rw/cstring.h>             // Rogue Wave RWCString
...
if (!plat.when("RAW_EVENT", Callback(raw_cb,0))) {
    cout << "Could not set raw callback" << endl;
}
...
void raw_cb( Ptr, Ptr calldata)
{
    // Extract information contained in the event
    CurrentEvent ce(calldata);
    RWCString event_type(ce.get_event().chp());
    RWCString distinguished_name(ce.get_objname().chp());
    RWCString event_class(ce.get_objclass().chp());
    RWCString event_info(ce.get_info().chp());

    //Process the information extracted
    if (event_type.contains("objectCreation")) {
        if (event_class.contains("dish")) {
            // Check the vchipId of the dish object against the
            // current channel. If the user isn't allowed
            // to watch this program, delete the user.
            //
            cout << "Checking permission..." << endl;
            if (!view_permission(distinguished_name)) {
                cout << "Not authorized to view program!" << endl;
                // Delete the dish object
                delete_object(distinguished_name);
            } else {
                cout << "Enjoy the show!" << endl;
            }
        }
    }
}
```

When the `raw_cb` callback defined in this example is called, the application checks the user's permissions to watch the current channel based on the following information extracted from the event:

- The entire text of the event, from which the event type is extracted
- The FDN of the managed object that generated the event
- The class of the managed object that generated the event

7.2.3 Controlling Event-Related Updates

A managed object sends an event notification to inform a management application of a change that has occurred to the managed object. The managed object itself is in an MIS or an agent in the network. To ensure that your application has access to current data about managed resources, the changes need to be propagated to `Image` and `Album` instances cached in your application.

Depending on your requirements, you can choose how changes to managed objects are propagated to `Image` and `Album` instances cached in your application.

7.2.3.1 Automatically Tracking Changes

Automatically tracking changes causes the MIS to update `Image` and `Album` instances cached in your application whenever the MIS receives an event notification.

To track changes automatically, set the `TRACKMODE` property for `Image` and `Album` instances as described in:

- Section 5.9.1 “Automatically Tracking Changes to an Object” on page 5-26 for `Image` instances
- Section 6.4.1.1 “Automatically Maintaining the Membership of an Object Collection” on page 6-13 for `Album` instances

7.2.3.2 Overriding Automatic Updates for a Specific Event Type

Overriding automatic updates for a specific event type prevents the MIS from updating `Image` and `Album` instances cached in your application whenever the MIS receives an event notification of the specified type. By default, `Image` and `Album` instances cached in your application are updated each time an event is received if you track changes to managed objects automatically.

Override automatic updates for a specific event type if:

- You want your application to process the previous values of attributes after an event of the type has been received.
- You want to improve the performance of your application by reducing the traffic between your application and the MIS.

To override automatic updates for specific event types, call the `do_nothing` function of the `CurrentEvent` class. Include the call to `do_nothing` in the callback function for handling the type of event. When you override automatic updates, include code in your callback to process the old values.

7.2.3.3

Tracking Changes From Within a Callback

Tracking changes from within a callback updates `Image` and `Album` instances cached in your application when the callback is executed. Track changes from within a callback if:

- **You want to track changes manually.** Manually tracking changes gives you control over which changes you update `Image` and `Album` instances with, and enhances the performance of your application.
- **You want to pre-empt automatic updates for a specific event type.** Pre-empting automatic updates enables you to access the updated values updates within the callback. By default, automatically tracked `Image` and `Album` instances are updated only after the callback has returned control to the application.

To track changes from within a callback, call the `do_something` function of the `CurrentEvent` class. Include the call to `do_something` in the callback function for handling the type of event. When you track changes from within a callback, include code in your callback to process the new values.

Code for tracking changes from within a callback is shown in CODE EXAMPLE 7-5.

CODE EXAMPLE 7-5 Tracking Changes From Within a Callback

```
...
#include <pmi/hi.hh>           // High Level PMI
...
Album channels;               // Collection of channel objects
Album dishes;                 // Collection of dish objects
...
void remove_cb( Ptr, Ptr calldata){
    CurrentEvent ce(calldata);
    ce.do_something();
    cout << "*****Removing from the Album ";
    RWCString event_class(ce.get_objclass().chp());
    Image tmpimage(ce.get_objname());
    if (event_class.contains("dish")) {
        cout << "dishes*****" << endl;
        dishes.exclude(tmpimage);
    } else if (event_class.contains("channel")) {
        cout << "channels*****" << endl;
        channels.exclude(tmpimage);
    }
}
...
```

In this example, `Image` and `Album` instances cached in the application are updated before the `exclude` function of the `Album` class is called. Updating the instances before calling `exclude` ensures that updated values are processed within the `remove_cb` callback.

7.3 Scheduling Event Handling

An application cannot predict when it will receive an event notification from a managed object. Therefore, to process an event notification, an application must be able to interrupt whatever function is being performed and process the event notification when it is received. To enable an application to interrupt the function it is currently performing, you must add a scheduler to your application. A scheduler retrieves each event from the event queue and calls the function required to process the event correctly.

Solstice EM provides the following schedulers:

- `sched` - For applications without a graphical user interface
- `xtsched` - For applications with a graphical user interface

If you want to develop your own scheduler, follow the guidelines given in Section 7.3.3 “Guidelines for Developing Your Own Scheduler” on page 7-15.

7.3.1 Scheduling for Applications Without a Graphical User Interface

To schedule event handling for an application without a graphical user interface, use the `sched` scheduler. To use the `sched` scheduler, call its `dispatch_recursive` function after all initialization is finished.

When you call the `dispatch_recursive` function, you have to specify whether the `sched` scheduler waits before processing an event. To specify this, set the boolean parameter of `dispatch_recursive` to:

- `FALSE` - Do not wait before processing events, then return
- `TRUE` - Wait for an event to be received, process the event, then return

To ensure that your application is able to receive events when they arrive, call the `dispatch_recursive` function in a loop. If you do not call `dispatch_recursive` in a loop, it is called only once. Any incoming events that arrive after the call to `dispatch_recursive` are not processed.

Note – It is typically more efficient to call `dispatch_recursive` with the parameter set to `TRUE`. Your application uses fewer CPU cycles while executing in the loop if you set the parameter to `TRUE` than if you set it to `FALSE`. If you set the parameter to `FALSE`, your application loops continually.

CODE EXAMPLE 7-6 shows a call to the `dispatch_recursive` function.

CODE EXAMPLE 7-6 Calling the `dispatch_recursive` Function

```
...
#include <pmi/hi.hh>      // High Level PMI
#include <pmi/sched.hh>   // Scheduler for applications with no GUI
...
while(MIS_Connected) {
    dispatch_recursive(TRUE);
}
...
```

In this example, the `dispatch_recursive` function is called while the application remains connected to an MIS.

When the `dispatch_recursive` function is called, the `sched` scheduler performs the following actions:

- Checks the event queue for events
- Retrieves each event from the queue
- Identifies the correct callback function for each event retrieved
- Calls the required callback function for each event, passing the event data to it as an argument

To call the `dispatch_recursive` function in an infinite loop by using a single line of code, use the `dispatch_main_loop` function. A call to `dispatch_main_loop` is shown in CODE EXAMPLE 7-7.

CODE EXAMPLE 7-7 Calling the `dispatch_main_loop` Function

```
...
#include <pmi/sched.hh>   // Scheduler for applications with no GUI
...
dispatch_main_loop();
...
```


The `dispatch_main_loop` function contains the code in CODE EXAMPLE 7-8.

CODE EXAMPLE 7-8 Contents of the `dispatch_main_loop` Function

```
...
while(TRUE) {
    dispatch_recursive(TRUE);
}
...
```

7.3.2 Scheduling for Applications With a Graphical User Interface

A graphical application must be able to process two types of events:

- Event notifications from managed objects
- X events resulting from user interaction with the application

To schedule the handling of both types of events, you must use the `xtsched` scheduler. The `xtsched` scheduler performs all the functions of the `sched` scheduler and handles all X events. Using the `xtsched` scheduler involves:

- Initializing and activating the `xtsched` scheduler
- Preventing erroneous user input

7.3.2.1 Initializing and Activating the `xtsched` Scheduler

Initializing the `xtsched` scheduler sets the application context. Every graphical application based on the X Window system requires an application context. After initializing the `xtsched` scheduler, activate it.

To initialize the `xtsched` scheduler, call the `xtsched` function `set_app_context`. The `set_app_context` routine passes the application context to the `xtsched` scheduler to enable the PMI to control X events.

To activate the `xtsched` scheduler, call its `XtAppMainLoop` function after all initialization is finished.

Code for initializing and activating the `xtsched` scheduler is shown in CODE EXAMPLE 7-9.

CODE EXAMPLE 7-9 Initializing and Activating the `xtsched` Scheduler

```
...  
#include <pmi/xtsched.hh>                // PMI GUI scheduler  
...  
// X-Windows scheduler initialization  
XtToolkitInitialize();  
app_context = XtCreateApplicationContext();  
set_app_context(app_context); // For xtsched  
...  
XtAppMainLoop(app_context);  
...
```

7.3.2.2 Preventing Erroneous User Input

An application that has to process data from incoming events may fail to respond to user input. In such cases, users often repeat mouse clicks or keystrokes, resulting in erroneous input. By designing your graphical applications to prevent erroneous user input, you can improve their usability. Prevent erroneous user input while an application is processing data by:

- **Giving feedback to the user.** Indicate to the user that the application is busy by changing the pointer to a watch or an hour glass.
- **Blocking user input to the application.** Use the `set_X_event_processing` function of the `xtsched` scheduler to disable the processing of X events. The `set_X_event_processing` function accepts a boolean parameter:
 - `FALSE` - Disables the processing of X events, thereby blocking user input
 - `TRUE` - Enables processing of X events, thereby enabling the application to accept user input

Note – If you disable the processing of X events, you must call `set_X_event_processing(TRUE)` to enable your application to process them again.

Code for disabling and enabling the processing of X events is shown in CODE EXAMPLE 7-10.

CODE EXAMPLE 7-10 Disabling and Enabling the Processing of X Events

```
...
#include <pmi/hi.hh>                // High Level PMI
#include <pmi/xtsched.hh>           // PMI GUI scheduler
...
void delete_button_press_callback(Widget w, XtPointer, XtPointer)
{
    // The button was pressed.
    // Disable processing of events
    set_X_event_processing(FALSE);
    ...
    Image im;
    ...
    im.delete();

    // Remove item
    ...
    // Now item is removed, permit the user to select the next item
    set_X_event_processing(TRUE);
}...
```

In this example, when the user selects an item and clicks the Delete button, all further user input is blocked until the item is removed. By blocking user input until the item is removed, the application prevents the user from attempting to delete an object that does not exist, thereby avoiding undesirable effects (for example, an error message or a failure of the application).

7.3.3 Guidelines for Developing Your Own Scheduler

The purpose of a scheduler is to enable an application to respond to events. A scheduler transfers control from the function an application is currently performing to the callback function registered to process the event.

Any scheduler that you develop must contain code that listens on all used file descriptors. In Solstice EM, whenever a callback is registered internally by the PMI or by an application, a file descriptor is used. When an event is received, the file descriptor is set, which causes the PMI to transfer control to the specified callback routine. The schedulers supplied with EM use the `select()` system call to check for any activity on the open file descriptors.

7.4 Filtering Events

Filtering events makes sure that the MIS only forwards to your application the events that are relevant to the application. Filter events to reduce the amount of network traffic between your application and the MIS it is connected to. By default, the MIS forwards all events it receives to your application.

Depending on the sources of events you want your application to receive, you can filter events by:

- Selecting managed object classes and event types
- Selecting a subtree of the management information tree (MIT)
- Specifying a discriminator construct

7.4.1 Selecting Managed Object Classes and Event Types

If you want your application to receive only events of particular types emitted by instances of particular managed object classes, select those managed object classes and event types. To filter events by selecting managed object classes and event types, call the `replace_discriminator_classes` function of the `Platform` class.

In the call to `replace_discriminator_classes` you must specify:

- A list of the managed object classes
- A list of event types

You must pass the lists of managed object classes and event types to the `replace_discriminator_classes` function in an array, even if there is only one managed object class or event type in the list.

Code for filtering events by selecting managed object classes and event types is shown in CODE EXAMPLE 7-11.

CODE EXAMPLE 7-11 Selecting Managed Object Classes and Event Types

```
...
#include <pmi/hi.hh>                                // High Level PMI
...
if (!plat.connect(server, "platform")) {
    cout << "Could not connect to EM server: ";
    cout << server << endl;
    return NOT_OK;
}
```

```
}  
...  
Array(DU) oc(3);  
Array(DU) event(1);  
oc[0] = "satellite";  
oc[1] = "channel";  
oc[2] = "dish";  
event[0] = "communicationsAlarm";  
plat.replace_discriminator_classes(oc,event);  
...
```

In this example, the MIS forwards only `communicationsAlarm` events emitted by instances of the `satellite`, `channel`, and `dish` managed object classes. All other events received by the MIS are not forwarded to the application.

Note – To be certain that your application receives all the events it is interested in, specify only managed object classes in the call to `replace_discriminator_classes`. If you specify no event types, all events emitted by instances of the specified managed object classes are forwarded.

7.4.2 Selecting a Subtree of the MIT

If you want your application to receive events only from managed objects in a particular subtree of the MIT, select the subtree. To filter events by selecting a subtree of the MIT, call the `replace_discriminator` function of the `Platform` class.

In the call to `replace_discriminator`, you have to specify the subtree by specifying:

- **Base managed object.** The base managed object specifies the FDN of the object that is the root of the subtree you want to select.
- **Scope.** The scope specifies which objects contained by the base managed object are selected.

Note – The `replace_discriminator` function does not allow you to apply a filter to a subtree.

Possible values of the scope in a subtree for the `replace_discriminator` function are given in TABLE 7-4.

TABLE 7-4 Scope Values in a Subtree for the `replace_discriminator` Function

Value	Meaning
0	Only events from the base managed object are received.
1	Only events from first-level subordinates of the base managed object are received.
2	Events from the base managed object and its entire subtree are received.
<code>individualLevels : n</code>	Only events from level <i>n</i> subordinates of the base managed object are received, where <i>n</i> is an integer.
<code>baseToNthLevel : n</code>	Events from the base managed object and all its subordinates to level <i>n</i> are received, where <i>n</i> is an integer.

To specify a subtree, construct a filter that tests that the value of the "DNFILTER" : `emDnScope` attribute equals the base managed object and the scope. The ASN.1 type definition of the "DNFILTER" : `emDnScope` attribute specifies that it is a SEQUENCE containing a base managed object and a scope.

Code for filtering events by selecting a subtree of the MIT is shown in CODE EXAMPLE 7-12.

CODE EXAMPLE 7-12 Selecting a Subtree of the MIT

```
...
#include <pmi/hi.hh>                                // High Level PMI
...
char discr[512];

strcpy(discr, "item : equality : {
    attributeId globalForm : \"DNFILTER\":emDnScope, attributeValue {
        base distinguishedName : {
            {
                {
                    attributeId \"Rec. X.721 | ISO/IEC 10165-2 :
                    1992\":systemId,
                    attributeValue name : \"wisconsin\"
                }
            }
        }
    } ,scope 2
}
```

CODE EXAMPLE 7-12 Selecting a Subtree of the MIT

```
");  
if (!plat.replace_discriminator(DU(discr))) {  
    cout << plat.get_error_string() << endl;  
    exit(3);  
}  
...
```

In this example, the MIS forwards only events emitted by the managed object `"/systemId=wisconsin"` and all managed objects in its entire subtree.

7.4.3 Specifying a Discriminator Construct

If you want your application to receive events selected according to the values of attributes in the events, specify a discriminator construct. Specifying a discriminator construct enables you to select events according to the values of the following attributes of an event:

- Managed object class
- Managed object instance
- Event type
- Any attribute specific to an event type, for example, for fault-related events:
 - Severity
 - Backed-up status
 - Probable cause

To filter events by specifying a discriminator construct, call the `replace_discriminator` function of the `Platform` class.

In the call to `replace_discriminator`, you have to specify the discriminator construct. The format of a discriminator construct is identical to the format of a filter in a derivation string. This format is defined in Section 6.3.2.3 “Filter” on page 6-7.

Note – Section 6.3.2.3 “Filter” on page 6-7 states that a filter in a derivation string is optional. This statement does not apply to a discriminator construct. Additionally, a discriminator construct *cannot* be combined with a base managed object and a scope.

Code for filtering events by specifying a discriminator construct is shown in CODE EXAMPLE 7-13.

CODE EXAMPLE 7-13 Specifying a Discriminator Construct

```
...
#include <pmi/hi.hh>                                // High Level PMI
...
char discrim [512];

strcpy(discrim, "CMISFilter(
    and: {
        item: equality: {probableCause, corruptData},
        item: equality: {perceivedSeverity, critical}
    }
)");

if (!plat.replace_discriminator(DU(discrim))) {
    cout << plat.get_error_string() << endl;
    exit(3);
}
...
```

In this example, the MIS forwards only events that have the value `corruptData` for `probableCause` and have the value `critical` for `perceivedSeverity`.

7.5 Simulating an Event

In a live network, events are generated as a result of activity on the network. If you want to test event handling by your application before it manages a live network, generate events for test purposes.

Depending on the type of event, and the rate at which you want to send events, you can simulate events without using the Solstice EM APIs, or you can simulate them programmatically.

7.5.1 Simulating an Event Without Using the Solstice EM APIs

You can generate events for test purposes by simulating an agent object, and creating and modifying objects in the MIS to simulate activity on a network. For more information, see Section 5.11 “Simulating an Agent Object” on page 5-30.

If you do not want to simulate an agent, use request templates to generate events. For more information on how to use request templates, refer to the *Customizing Guide*.

7.5.2 Simulating an Event Programmatically

To simulate an event programmatically, write an application that is acting in an agent role. Simulating an event programmatically involves:

- Creating and initializing an instance of the `Image` class to represent the managed object that is emitting the event
- Requesting the event to be sent
- Sending the event to the MIS

For information on how to create and initialize an instance of `Image`, refer to Section 5.2.1 “Creating and Initializing an Instance of `Image`” on page 5-3.

To request an event to be sent, call the `send_event` function of the `Image` class. In the call to `send_event`, specify:

- **The event type.** The event type is defined in the object model of the managed system. It must be one of the event types supported by the managed object.
- **The event information.** The event information consists of a list of attributes and the values you want to set them to in the event. The list of attributes must be as defined in the GDMO specification of the event type.

To send an event to the MIS, call one of the following schedulers as described in Section 7.3 “Scheduling Event Handling” on page 7-11:

- `sched` - For applications without a graphical user interface
- `xtsched` - For applications with a graphical user interface

Note – The `send_event` function is a blocking function and does not send an event to the MIS. To send the event, you must call a scheduler after calling the `send_event` function. Otherwise, the event will be lost.

Code for simulating an event programmatically is shown in CODE EXAMPLE 7-14.

CODE EXAMPLE 7-14 Simulating an Event Programatically

```
...
#include <pmi/hi.hh>                                // High Level PMI
#include <pmi/xtsched.hh>                            // PMI GUI scheduler
#include <rw/cstring.h>                              // Rogue Wave RWCString
...
RWCString fdn, object_class;
...
Image im((char *)fdn.data(), (char*)object_class.data());
char event[30] = "communicationsAlarm";
char event_info[100] = "{ probableCause localValue : 85,"
                      "perceivedSeverity critical }";
if (im.get_error_type() != PMI_SUCCESS) {
    cout << im.get_error_string() << endl;
    exit(5);
}
// Request the event to be sent
if (!im.send_event(event, event_info)) {
    cout << "Error sending event: ";
    cout << im.get_error_string() << endl;
    exit (6);
}
...
// Send the event to the MIS
dispatch_recursive(TRUE);
...
```

In this code example, a `communicationsAlarm` event is sent to the MIS. The `probableCause` attribute is set to a locally defined value of 85, and the `perceivedSeverity` attribute is set to `critical`.

The variable `fdn` specifies the FDN of a managed object (for example `"/systemId=\"mazes\"/satelliteId=\"NorthernLights\""`). The variable `object_class` specifies the name of a managed object class (for example the `satellite` class from the sample programs). The initialization of the `fdn` and `object_class` variables is not shown in the example.

7.6 Subscribing to Log Record Events

An application receives log record events only if it has subscribed to them. Log record events are not sent to an application unless the application has subscribed to them.

Subscribing to log record events involves:

- Selecting the managed object that represents the application's connection to the MIS
- Setting the `emSpecialEvents` attribute of the managed object to `logRecordEvent`

For information on how to select a managed object, refer to Section 5.3 “Selecting a Managed Object” on page 5-8. To obtain the FDN of the managed object that represents the application's connection to the MIS, call the `get_prop` function of the `Platform` class. In the call to `get_prop`, specify the `APPLICATION_OBJNAME` property.

For information on how to set an attribute of a managed object, refer to Section 5.7 “Setting Attribute Values of an Object” on page 5-19. You have to set the `emSpecialEvents` attribute to the text string `logRecordEvent`. Therefore, call the `set_str` function of the `Image` class to set this attribute in the `Image` instance.

Code for subscribing to log record events is shown in CODE EXAMPLE 7-15.

CODE EXAMPLE 7-15 Subscribing to Log Record Events

```
...
#include <pmi/hi.hh>                                // High Level PMI
...
Platform em(duEM);
...
// Get application instance
DU appinst = em.get_prop(duAPPLICATION_OBJNAME) ;

// Construct the Image object
Image app_image(appinst);
if(!app_image.boot()) {
    cout << app_image.get_error_string() << endl;
}
...
```

CODE EXAMPLE 7-15 Subscribing to Log Record Events (*Continued*)

```
// Subscribe to log record events
if(!app_image.set_str("emSpecialEvents",{logRecordEvent})) {
    strcpy(pmi_error_msg, app_image.get_error_string());
    app_image.reset_error();
}

// Store the changes
if(!app_image.store()){
    strcpy(pmi_error_msg, app_image.get_error_string());
    app_image.reset_error();
}
...
```

Performing Asynchronous Management Operations

A management operation can take a significant length of time to finish, particularly if the operation exchanges a large quantity of data between your application and the network resources it is managing. If your application is blocked while waiting for an operation to finish, the application may appear unresponsive to a user, or may fail to respond quickly enough to important events on your network. Performing asynchronous management operations enables an application to continue with other processing without waiting for the operations to finish.

This chapter explains how to enable applications to perform asynchronous management operations.

- Section 8.1 “Asynchronous and Synchronous Operation” on page 8-1
- Section 8.2 “Specifying Asynchronous Operations” on page 8-2
- Section 8.3 “Handling Responses From an Asynchronous Operation” on page 8-11
- Section 8.4 “Verifying and Changing the Status of an Asynchronous Operation” on page 8-23

8.1 Asynchronous and Synchronous Operation

Your application can perform a management operation in synchronous or asynchronous mode.

In *synchronous mode*, all functions within the application wait until the operation is complete before returning. In this way, the thread of control is blocked after a function is called, and the application can make use of the result immediately after the function returns.

In *asynchronous mode* an application can initiate several concurrent asynchronous operations before receiving any of the results. The results are not guaranteed to be returned in any particular order. Asynchronous mode enables an application to continue with other processing without waiting for a function to return, but does not enable the result to be used immediately.

Note – Despite the similarities in terminology, these modes are not related to synchronization.

8.2 Specifying Asynchronous Operations

To enable an application to continue processing while the application is waiting for the result of an operation, call the asynchronous function for the operation. Calling an asynchronous function initiates an operation, but returns program control to your application before the operation finishes.

In the call to an asynchronous function, you can optionally specify a callback function to be run when the asynchronous operation finishes. For more information, see Section 8.3 “Handling Responses From an Asynchronous Operation” on page 8-11.

An asynchronous function returns an instance of the `Waiter` class. The function does not return the result of the operation initiated. The `Waiter` class represents an unfinished asynchronous operation. The instance of `Waiter` returned enables you to:

- Register a callback function for handling response from managed objects as described in Section 8.3.2 “Registering a Callback Function for Handling Responses From Managed Objects” on page 8-12
- Verify or change the status of an asynchronous operation as described in Section 8.4 “Verifying and Changing the Status of an Asynchronous Operation” on page 8-23.

You can choose to perform the following types of operations asynchronously:

- Interactions with the MIS
- Operations on managed objects
- Operations on object collections

In addition, the `Album` class provides functions for performing asynchronous CMIS operations on object collections.

8.2.1 Interactions With the MIS

The `Platform` class provides functions for synchronously or asynchronously interacting with the MIS. The names of asynchronous functions are prefixed with `start`. To specify whether your application interacts with the MIS synchronously or asynchronously, call the appropriate version of the function as listed in TABLE 8-1.

TABLE 8-1 Synchronous and Asynchronous Functions of the `Platform` Class

Operation	Synchronous	Asynchronous
Connect an application to the MIS	<code>connect</code>	<code>start_connect</code>
Disconnect an application from the MIS	<code>disconnect</code>	<code>start_disconnect</code>

8.2.2 Asynchronous Operations on Managed Objects

For each management operation that you can perform on a managed object, the `Image` class provides a function for performing it synchronously or asynchronously. The names of asynchronous functions are prefixed with `start`. To specify whether a management operation is performed synchronously or asynchronously, call the appropriate version of the function as listed in TABLE 8-2.

TABLE 8-2 Synchronous and Asynchronous Functions of the `Image` Class

Operation	Synchronous	Asynchronous
Activate or update an instance of <code>Image</code>	<code>boot</code>	<code>start_boot</code>
Deactivate an instance of <code>Image</code>	<code>shutdown</code>	<code>start_shutdown</code>
Update attribute values in the MIS	<code>store</code>	<code>start_store</code>
Add a managed object to the MIS	<code>create</code>	<code>start_create</code>
Add, in containing object, a managed object to the MIS	<code>create_within</code>	<code>start_create_within</code>
Remove a managed object from the MIS	<code>destroy</code>	<code>start_destroy</code>
Send an action request with a text parameter	<code>call</code>	<code>start</code>
Send an action request with an encoded parameter	<code>call_raw</code>	<code>start_raw</code>

8.2.3 Asynchronous Operations on Object Collections

For each management operation that you can perform on an object collection, the `Album` class provides a function for performing it synchronously or asynchronously. The names of asynchronous functions contain the text `start`. To specify whether a management operation is performed synchronously or asynchronously, call the appropriate version of the function as listed in TABLE 8-3.

TABLE 8-3 Synchronous and Asynchronous Functions of the `Album` Class

Operation	Synchronous	Asynchronous
Define the membership of an object collection	<code>derive</code>	<code>start_derive</code>
Activate all <code>Image</code> instances in an object collection	<code>all_boot</code>	<code>all_start_boot</code>
Deactivate all <code>Image</code> instances in an object collection	<code>all_shutdown</code>	<code>all_start_shutdown</code>
Update attribute values in the MIS for all <code>Image</code> instances in an object collection	<code>all_store</code>	<code>all_start_store</code>
Add all managed objects in an object collection to the MIS	<code>all_create</code>	<code>all_start_create</code>
Add, in a containing object, all managed objects in an object collection to the MIS	<code>all_create_within</code>	<code>all_start_create_within</code>
Remove all managed objects in an object collection from the MIS	<code>all_destroy</code>	<code>all_start_destroy</code>
Send an action request with a text parameter to all managed objects in an object collection	<code>all_call</code>	<code>all_start</code>
Send an action request with an encoded parameter to all managed objects in an object collection	-	<code>all_start_raw</code>

Note – The `Album` class does not provide a function for sending an encoded synchronous action request to an object collection.

8.2.4 Asynchronous CMIS Operations on Object Collections

The `Album` class provides functions for performing asynchronous management operations defined by the common management information service (CMIS). CMIS is specified in recommendation ITU-T X.710/ISO-9595 *Common Management Information Services (CMISE)*. Use these functions if you want to perform an operation on a subset of the managed objects in an object collection.

You can perform asynchronous management operations on an object collection only if all of the following conditions are met:

- The membership of the object collection is defined by derivation (see Section 6.3.1 “Defining the Membership by Derivation” on page 6-3).
- The managed objects that you want to perform the operation on are all contained in a subtree of the MIT rooted in the base managed object of the object collection.
- The managed objects in the object collection reside in an agent that supports the common management information protocol (CMIP).

Note – The `Album` class also provides functions for performing management operations that are independent of any particular management protocol or service. However, these functions allow you to perform a management operation only on all managed objects in an object collection, not a subset of them. For more information, refer to Section 6.5 “Accessing All Objects in an Object Collection” on page 6-17.

The `Album` class provides functions for performing the CMIS operations listed in TABLE 8-4. For a definition of all CMIS operations, refer to ITU-T X.710/ISO-9595 *Common Management Information Services (CMISE)*.

TABLE 8-4 CMIS Operations Supported by the `Album` Class

Operation	Definition
M-DELETE	Deletes a managed object
M-GET	Obtains attribute values from a managed object
M-SET	Modifies attribute values for a managed object
M-ACTION	Performs an action on a managed object

Performing an asynchronous CMIS operation on an object collection involves:

- Selecting the managed objects to be the subject of a CMIS operation
- Requesting a CMIS operation

8.2.4.1 Selecting the Managed Objects to be the Subject of a CMIS Operation

The derivation string of the Album instance that contains an object collection selects the managed objects that are the subject of a CMIS operation on the object collection.

If you want to select all managed objects in an object collection, leave the derivation string unchanged after you call the derive function of the Album class.

If you want to select a subset of the managed objects in an object collection, reset the derivation string but do not call the derive function of the Album class.

Code for selecting the managed objects to be the subject of a CMIS operation is shown in CODE EXAMPLE 8-1.

CODE EXAMPLE 8-1 Selecting Managed Objects for a CMIS Operation

```
...
#include <pmi/hi.hh>           // High Level PMI
...
Album nologs_album = Album("All objects except log objects");
...
// Set derivation string for populating the object collection
// and then start the derivation

DU ALL_OBJECTS_EXCEPT_LOG =
    "ALL/CMISFilter(not:{item:equality:{objectClass,log}})";

if (!nologs_album.set_derivation(ALL_OBJECTS_EXCEPT_LOG)) {
    cout << nologs_album.get_error_string() << endl;
    exit(3);
}

if (!nologs_album.derive()) {
    cout << nologs_album.get_error_string() << endl;
    exit(4);
}

...
// Reset the derivation string to select objects for CMIS operation,
// but do not start a derivation
DU ALL_SECOND_LEVEL_OBJECTS_EXCEPT_LOG =
    "LV(2)/CMISFilter(not:{item:equality:{objectClass,log}})";
if (!nologs_album.set_derivation(ALL_SECOND_LEVEL_OBJECTS_EXCEPT_LOG)) {
    cout << nologs_album.get_error_string() << endl;
    exit(5);
}
...
```

In this example, an object collection containing all managed objects except instances of the `log` class is created. After the membership of this object collection has been defined by derivation, the derivation string is reset to select all second-level subordinates of the `system` object except instances of the `log` class. To preserve the membership of the object collection, the `derive` function of the `Album` class is not called after the derivation string is reset.

8.2.4.2 Requesting a CMIS Operation

When you have selected the objects to be the subject of a CMIS operation, call one of the functions listed in TABLE 8-5 to request the operation. Additional information on how to call each function listed in TABLE 8-5 is given in the following subsections.

TABLE 8-5 Functions of the `Album` Class for Requesting CMIS Operations

Operation	Function
M-GET	<code>start_m_get</code>
M-SET	<code>start_m_set</code>
M-ACTION with a text parameter	<code>start_m_action</code>
M-ACTION with an encoded parameter	<code>start_m_action_raw</code>
M-DELETE	<code>start_m_delete</code>

Requesting an Asynchronous CMIS M-GET Operation

To request an asynchronous CMIS M-GET operation, call the `start_m_get` function of the `Album` class.

In the call to `start_m_get`, specify:

- A list of the attributes you want to get the values of. You must pass the list of attributes to the `start_m_get` function in an array, even if you want to get only one attribute.
- A callback function to be called when the management operation is complete.

Requesting an Asynchronous CMIS M-SET Operation

To request an asynchronous CMIS M-SET operation, call the `start_m_set` function of the `Album` class.

In the call to `start_m_set`, specify:

- A modification list. The modification list specifies the attributes you want to set and how you want to set them.
- A callback function to be called when the management operation is complete.

Note – The `start_m_set` function supports only best effort synchronization.

To specify a modification list, create a queue of `AttrModifier` instances and pass this queue to `start_m_set`. The `AttrModifier` class represents a single modification to an attribute of a managed object.

Creating a queue of `AttrModifier` instances involves:

- Initializing an instance of the `Queue` class
- For each modification in the modification list:
 - Initializing an instance of the `AttrModifier` class
 - Calling functions of the `AttrModifier` to specify how you want to set the attribute
 - Calling the `enq` function of the `Queue` class to add the modification to the queue

In the call to the constructor of the `AttrModifier` class, specify the name of the attribute you want to set.

To specify how you want to set the attribute, call functions of the `AttrModifier` class as follows:

- To specify the value that you want to set the attribute to, call the `set_value` function. In the call to `set_value`, you have to specify an instance of the `Morf` class that represents the value. For information on the `Morf` class, refer to Chapter 9.
- To specify how the attribute value is to be modified, call the `set_operator` function. In the call to `set_operator`, you have to specify one of the operations listed in TABLE 8-6.

TABLE 8-6 Operations for the `set_operator` Function

Operation	Result
REPLACE	Replaces the existing value with that specified in the function call. It corresponds to the REPLACE operation in a property list. This is the default operation.
ADD	Adds the value specified in the function call to the current value of a multi-valued attribute. It corresponds to the ADD operation in a property list. Specify the ADD operation for multi-valued attributes only. If you specify the ADD operation for a single-valued attribute, an exception is thrown.
REMOVE	Removes the value specified in the function call from the current value of a multi-valued attribute. It corresponds to the REMOVE operation in a property list. Specify the REMOVE operation for multi-valued attributes only. If you specify the REMOVE operation for a single-valued attribute, an exception is thrown.
SET_TO_DEFAULT	Replaces the existing value with the default value defined in the property list in the ATTRIBUTES construct of the attribute's GDMO specification.

Code for requesting an asynchronous CMIS M-SET operation is shown in CODE EXAMPLE 8-2.

CODE EXAMPLE 8-2 Requesting an Asynchronous CMIS M-SET Operation

```
...
#include <pmi/hi.hh>                // High Level PMI
#include <queue.hh>                 // Queue class declaration
...
Album nologs_album = Album("All objects except log objects");
...
Queue (AttrModifier) amq;
Morf val;
AttrModifier * elt1 = new AttrModifier (CDU coordinates);
elt1->set_operator (REPLACE);
elt1->set_value (&val);
amq.enq (elt1);
AttrModifier * elt2 = new AttrModifier (CDU highWaterMark);
elt2->set_operator (SET_TO_DEFAULT);
amq.enq (elt2);
nologs_album.start_m_set (amq);
...
```

In this example, a queue named `amq` of `AttrModifier` instances is created. The `AttrModifier` instances in the queue specify that:

- Attribute `coordinates` is set to the value specified in the instance of `Morf` named `val`.
- Attribute `highWaterMark` is set to its default value.

Requesting an Asynchronous CMIS M-ACTION Operation With a Text Parameter

To request an asynchronous CMIS M-ACTION operation with a text parameter, call the `start_m_action` function of the `Album` class.

In the call to `start_m_action`, specify:

- The name of the action
- An instance of the `DataUnit` class that contains the parameter associated with the action
- A callback function to be called when the management operation is complete

Note – The `start_m_action` function supports only best effort synchronization.

Requesting an Asynchronous CMIS M-ACTION Operation With an Encoded Parameter

To request an asynchronous CMIS M-ACTION operation with an encoded parameter, call the `start_m_action_raw` function of the `Album` class.

In the call to `start_m_action_raw`, specify:

- The name of the action
- An instance of the `Morf` class that contains the parameter associated with the action
- A callback function to be called when the management operation is complete

Note – The `start_m_action_raw` function supports only best effort synchronization.

Requesting an Asynchronous CMIS M-DELETE Operation

To request an asynchronous CMIS M-DELETE operation, call the `start_m_delete` function of the `Album` class. In the call to `start_m_delete`, specify a callback function to be called when the management operation is complete.

8.3 Handling Responses From an Asynchronous Operation

An application receives the following types of responses from an asynchronous operation:

- **A confirmation that the operation has finished.** Some processes in your application may depend on the completion of an asynchronous operation, and must wait until the asynchronous operation has finished. For example, an application that connects asynchronously to an MIS must wait until the connection has been established before attempting to access managed objects in the MIS. Use the confirmation to make your application wait until an asynchronous operation has finished before it starts a process that depends on the completion of the asynchronous operation.
- **Responses from managed objects.** Each managed object that is the subject of an asynchronous operation sends a response that contains the result of the operation. Handle these responses to process the result of an asynchronous operation, for example by displaying retrieved values or taking alternative courses of action depending on whether the operation succeeds or fails.

Handling responses to asynchronous operations involves:

- Registering a callback function for the completion of an asynchronous operation
- Registering a callback function for handling responses from managed objects
- Writing a callback function
- Scheduling response handling

8.3.1 Registering a Callback Function for the Completion of an Asynchronous Operation

Register a callback to specify the processing that is carried out when an asynchronous operation finishes. Registering a callback associates the callback with an asynchronous operation. The callback is run when your application receives a confirmation that the asynchronous operation has finished.

Registering a callback function for the completion of an asynchronous operation is optional. By default, no callback is registered when you call a function for performing an asynchronous operation.

To register a callback, pass it as a parameter to the asynchronous function. In the asynchronous function call, initialize an instance of the `Callback` class to represent the callback function.

To initialize an instance of the `Callback` class, call its constructor in the asynchronous function call. In the call to the constructor of the `Callback` class, you must specify:

- The name of the callback function.
- A pointer to the data to be passed as an argument to the callback function. You can specify a null pointer if you do not want to pass any data to the callback function.

Code for registering a callback for an asynchronous operation is shown in CODE EXAMPLE 8-3.

CODE EXAMPLE 8-3 Registering a Callback for an Asynchronous Operation

```
...
#include <pmi/hi.hh>           // High Level PMI
...
Waiter alb_waiter = myAlbum.start_derive(
    Callback(alb_derived_cb, (Ptr)&cb_info));
...
```

In this example, the `start_derive` function of the `Album` class is called to start the derivation of an `Album` instance asynchronously. The callback function `alb_derived_cb` is registered so that it is called when the derivation finishes. A pointer to data contained in the `cb_info` object is passed to the callback function.

8.3.2 Registering a Callback Function for Handling Responses From Managed Objects

The callback registered in the call to a function for an asynchronous operation is called only once, when the operation is completed. To handle responses from managed objects to an asynchronous function call, you must register a callback specifically for that purpose.

To register a callback for handling responses from managed objects, call the `when_resp` function on the `Waiter` instance returned by the asynchronous function call.

In the call to `when_resp`, specify the callback function for handling responses from managed objects. This callback function is called each time your application receives a response to the asynchronous operation that returned the `Waiter` instance on which `when_resp` is called.

Code for registering a callback function to handle responses from managed objects is shown in CODE EXAMPLE 8-4.

CODE EXAMPLE 8-4 Registering a Callback Function for Response Handling

```
...
#include <pmi/hi.hh>           // High Level PMI
...
Waiter cur;
...
if (!(cur=test_album.all_start(DU("topoGetNodeReport"),
    DU("NULL"),Callback(done_cb, &done)))) {
    cout << test_album.get_error_string() << endl;
    exit(9);
}
if (cur.get_except()) {
    cout << cur.get_except()->reason() << endl;
    exit(10);
}
// subscribe to any future incoming replies
cur.when_resp(Callback(asyn_cb,0));
...
```

In this example, the `all_start` function of the `Album` class is called to send an asynchronous action request with a text parameter to all managed objects in an object collection. The `when_resp` function is called on the `Waiter` instance returned by the call to `all_start`. The call to `when_resp` registers a callback function named `asyn_cb`. The `asyn_cb` function is called each time the application receives a response to the asynchronous action from a managed object.

8.3.3 Writing Callback Functions for Asynchronous Operations

Write a callback function to handle each type of asynchronous operation and response you want to process. Each callback function you write must contain the code needed to carry out the processing you require. Writing a callback function involves:

- Defining the signature of the callback function
- Writing code for handling either of the following:
 - A confirmation that an asynchronous operation has finished
 - Responses from managed objects

8.3.3.1 Defining the Signature of the Callback Function

The signature of any callback function you write must be in the following format:

```
void callbackName (Ptr userdata, Ptr calldata)
```

Where:

- *callbackName* is the name you assign to your callback function.
- *userdata* is the data passed to the callback by your application.
- *calldata* is the data passed to the callback by the scheduler that calls the callback function.

Note – This format is identical to that required for event-handling callbacks as defined in Section 7.2.2 “Writing Callback Functions for Event Handling” on page 7-6.

8.3.3.2 Writing Code for Handling a Confirmation That an Asynchronous Operation Has Finished

In the body of your callback function, provide code that you want your application to run when an asynchronous operation finishes. The code you write depends on what you want the callback to do when it is run. If you do not need to process the data that the scheduler passes to the callback function, you can ignore callback’s second argument.

An example of a callback for processing a confirmation that an asynchronous operation has finished is given in CODE EXAMPLE 8-5.

CODE EXAMPLE 8-5 Callback for Completion of an Asynchronous Operation

```
...
#include <pmi/hi.hh>                // High Level PMI
...
void alb_derived_cb(Ptr context_data, Ptr)
{
    callback_data *cb_data = (callback_data *)context_data;
    char **args_list = cb_data->get_args();
    char *g_class = cb_data->get_derivation_class();
    fprintf(stdout,
        "\n~~~~~\n");
    fprintf(stdout,
        "%s: Album Derivation Callback Function\n",
```

```

        args_list[0]);
Album alb = cb_data->get_album();
if (!alb)
    fprintf(stderr,
        "%s: Error initializing Album in callback\n",
        args_list[0]);
else {
    fprintf(stdout,
        "Number of %s instances found: %d\n",
        g_class, alb.num_images());
}

fprintf(stdout,
    "\n~~~~~\n");
delete cb_data;
fprintf(stdout,
    "\n%s: Program complete, terminating MIS connection\n\n", args_list[0]);
MIS_Connected = FALSE;
}

```

When this callback is run, it checks for the success of the derivation. If the derivation fails, the callback prints a message stating that the derivation has failed. If the derivation succeeds, the callback prints the number of `Image` instances in the derived `Album` instance.

8.3.3.3 Writing Code for Handling Responses From Managed Objects

In the body of your callback function, provide code that you want your application to run when it receives a response from a managed object. The code you write depends on what you want the callback to do when it is run.

Making Correct Use of the Data Passed By the Scheduler

If you intend to register your callback by calling `when_resp`, you *must* use the data that the scheduler passes to the callback function to build a `CurrentEvent` object. This data is a void pointer. It is passed in the second argument of the callback.

If you ignore this data, memory allocated by the scheduler for this data is not freed, leading to a memory leak.

Code that correctly uses the data passed by the scheduler is shown in CODE EXAMPLE 8-6.

CODE EXAMPLE 8-6 Correct Use of Data Passed by the Scheduler

```
...
void cb(Ptr userdata, Ptr calldata)
{
    // do whatever
    if(calldata)
    {
        CurrentEvent ce(calldata);
        //      Do whatever and use and access the information
        //              within the CurrentEvent ce
    }
    // do whatever
}
...
```

In this example, the data passed by the scheduler is used in the callback to build a `CurrentEvent` object. The memory allocated by the scheduler for this data is freed, thereby avoiding a memory leak.

Code that does not correctly use the data passed by the scheduler is shown in CODE EXAMPLE 8-7.

CODE EXAMPLE 8-7 Incorrect Use of Data Passed by the Scheduler

```
...
void cb(Ptr userdata,Ptr calldata)
{
    // do whatever but never use calldata to build
    // a CurrentEvent object.

}
// or
void cb(Ptr userdata)
{
    // do whatever and ignore the second argument
}
...
```

In this example, the data passed by the scheduler is ignored, leading to a memory leak.

Extracting Information Contained In a Response From a Managed Object

Extract the information contained in responses that is useful to the user of your application. To extract information contained in a response, call functions of the `CurrentEvent` class in the body of your callback function.

The information available depends on the response.

Information available from all responses and the functions for extracting the information are given in TABLE 8-7.

TABLE 8-7 Information Available From All Responses

Information	Function
A pointer to the response that caused the callback function to be called. This pointer has no meaning outside the scope of the <code>CurrentEvent</code> object that represents the response.	<code>get_message</code>
The <code>Album</code> instance associated with the response.	<code>get_album</code>
The <code>Image</code> instance associated with the response.	<code>get_image</code>
The class of the managed object that sent the response.	<code>get_objclass</code>
The fully distinguished name (FDN) of the managed object that sent the response.	<code>get_objname</code>

Information available only from action replies sent in response to CMIS `M-ACTION` requests and the functions for extracting the information are given in TABLE 8-8.

TABLE 8-8 Information Available Only From Action Replies

Information	Function
The action reply information in encoded form.	<code>get_info_raw</code>
The name of the action request.	<code>get_eventtype</code>

Note – The information available from responses is a subset of the information available from event notifications. Functions of the `CurrentEvent` class that are not listed in TABLE 8-7 and TABLE 8-8 extract meaningful information only from event notifications, not from responses. For more information, refer to Section 7.2.2.2 “Extracting Information From an Event Notification” on page 7-7.

Example Callback for Handling Responses From Managed Objects

A callback function for handling responses from managed objects is shown in CODE EXAMPLE 8-8.

CODE EXAMPLE 8-8 Callback for Handling Responses From Managed Objects

```
...
#include <pmi/hi.hh>                // High Level PMI
...
void asyn_cb( Ptr , Ptr calldata)
{
    static int num = 1;
        cout << "\nExecuting asyn1 callback function for ";
        cout << num << " times";
        cout << endl;
        cout << "-----" << endl;
        num++;
        // Get and print the new attribute value.
        cout << "During the all_start operation ";
        cout << endl;
    if(calldata)
    {
        CurrentEvent ce(calldata);
        cout << "OBJNAME = " << ce.get_objname().chp() << endl;
        cout << "OBJCLASS = " << ce.get_objclass().chp() << endl;
        MessagePtr msg = (MessagePtr)ce.get_message();

        if(msg->type()==ACTION_RES)
        {
            ActionRes* srmsg = (ActionRes*)msg;
            cout << "OBJCLASS = " << oc2name(srmsg->oc).chp() << endl;
            cout << "FDN = " << oi2fdn(srmsg->oi).chp() << endl;
            cout << "ACTION-TYPE = " << endl;
            (srmsg->action_type).print(stdout);
            cout << "\n" << endl;
            cout << "ACTION-REPLY = " << endl;
            (srmsg->action_reply).print(stdout);
            cout << "\n" << endl;
        }

        Morf mf = ce.get_info_raw();
        Asn1Value val = mf.get_value();
        if(val)
        {
            cout << "info_raw() field of current event ACTION-REPLY = " << endl;
            val.print(stdout);
        }
    }
}
```

```

        cout << "\n" << endl;
    }
    cout << "eventtype() field of current event ACTION-TYPE = " <<
    ce.get_eventtype().chp() << endl;

    cout << "Information setting in the current event related Album " << endl;
    cout << "Derivation rule for the Album " <<
    ce.get_album().get_prop(duNICKNAME).chp() << " is : " <<
    ce.get_album().get_derivation().chp() << endl;
    cout << "\n" << endl;

    cout << "Information setting in the current event related Image" << endl;
    Image im = ce.get_image();
    cout << " image name is " << im.get_objname().chp() << endl;
    cout << " image class is " << im.get_objclass().chp() << endl;
    cout << " image state is " << im.get_state().chp() << endl;
    cout << " image last_error is " << im.get_last_error().chp() << endl;
    if (im.exists())
        cout << " image exists " << endl;
    else
        cout << " image does not exist " << endl;
    cout << " attribute(s) and attribute value(s) setting in the image " << endl;
    Array(DU) attr_names = im.get_attr_names();
    for (int i=0; i<attr_names.size; i++) {
        char *name = strdup(attr_names[i].chp());
        cout << name;
        cout << ": ";
        cout << im.get_str(name).chp() << endl;
    }
    cout << "\n" << endl;
}
}

```

The callback in this example checks a response to determine if the response is an action reply. If the response is an action reply, the following information about the action reply are printed:

- The managed object class of the action reply
- The FDN of the action reply
- The type of action request sent
- The text of the action reply message

For any response, the callback then prints out the following information:

- The nickname of the object collection to which the request was sent
- The derivation string of the object collection to which the request was sent
- The FDN of the managed object that sent the reply

- The managed object class of the managed object that sent the reply
- The state of the managed object that sent the reply
- The last error associated with the managed object that sent the reply

8.3.4 Scheduling Response Handling

An application cannot predict when it will receive a response to an asynchronous operation. Therefore, to process a response to an asynchronous operation, an application must be able to interrupt whatever function is being performed and process the response when it is received. To interrupt the function it is currently performing, an application requires a scheduler. A scheduler retrieves each event from the event queue and calls the function required to process the event correctly.

To schedule response handling, use one of the schedulers that Solstice EM provides:

- `sched` - For applications without a graphical user interface
- `xtsched` - For applications with a graphical user interface

Use these schedulers for response handling in the same way that you use them for event handling as explained in:

- Section 7.3.1 “Scheduling for Applications Without a Graphical User Interface” on page 7-11
- Section 7.3.2 “Scheduling for Applications With a Graphical User Interface” on page 7-13

When you use the `sched` scheduler for response handling, you can schedule response handling in either of the following modes, depending on the requirements of your application:

- **Nonblocking mode.** Schedule response handling in nonblocking mode if you want your application to continue with other operations while an asynchronous operation is outstanding.
- **Blocking mode.** Schedule response handling in blocking mode if you want your application to carry out some operations only after an asynchronous operation has finished. Blocking mode enables your application to carry out other operations while the asynchronous operation is outstanding.

When you use the `xtsched` scheduler for response handling, you can schedule response handling in nonblocking mode only.

8.3.4.1 Scheduling Response Handling in Nonblocking Mode

In nonblocking mode, your application continues with other operations while an asynchronous is outstanding.

To schedule response handling in nonblocking mode, activate the scheduler in a loop as shown in CODE EXAMPLE 8-9.

CODE EXAMPLE 8-9 Scheduling Nonblocking Asynchronous Response Handling

```
...
#include <pmi/hi.hh>           // High Level PMI
...
// Non-Blocking Example (note: boot_cb not shown in this example):
    Waiter waiter1 = cell_image.start_create(Callback(boot_cb, 0));
    waiter1.waitmore(3.0); // Specify 3 sec. timeout for boot operation
    while (MIS_Connected)
        dispatch_recursive(TRUE);
...
```

In this example, the `dispatch_recursive` function of the `sched` scheduler is called while the application remains connected to an MIS.

8.3.4.2 Scheduling Response Handling in Blocking Mode

In blocking mode, your application carries out some operations only after an asynchronous operation has finished. Blocking mode enables your application to carry out other operations while the asynchronous operation is outstanding.

To schedule response handling in blocking mode, activate the scheduler in a loop only while the asynchronous operation is still outstanding. To test whether the asynchronous operation is still outstanding, call the `was_completed` function of the `Waiter` class repeatedly until it returns a value indicating that the operation represented by the `Waiter` instance has finished.

Outside the loop, call functions for operations that you want to be carried out only after the asynchronous operation has finished.

Inside the loop, call functions for operations you want to be carried out while the asynchronous operation is outstanding. These operations must not block the application.

Code for scheduling response handling in blocking mode is shown in CODE EXAMPLE 8-10.

CODE EXAMPLE 8-10 Scheduling Blocking Asynchronous Response Handling

```
...
#include <pmi/hi.hh>      // High Level PMI
...
// Blocking Example:
    Waiter waiter2 = cell_album.all_start_store();
    while (!waiter2.was_completed){
        // Do operations you want to be carried out while the
        // asynchronous operation is outstanding. These operations
        // must not block the application.
        dispatch_recursive(TRUE);
    }
// Continue with operations you want to be carried out only after
// the asynchronous operation has finished.
...
```

In this example, attribute values in the MIS for all `Image` instances in an object collection are updated asynchronously. To ensure that the `dispatch_recursive` function of the `sched` scheduler is called only while this operation is still unfinished, it is called in a loop while the `was_completed` function returns `FALSE`.

8.3.5 Adding a Callback to the Scheduler Queue

Add a callback to the scheduler queue when you want the callback to be run. Adding a callback to the scheduler queue enables you to specify data that the scheduler passes to your callback when the callback is run.

You only need to add a callback to the scheduler queue if you are implementing an asynchronous function yourself. You do not need to add a callback to the scheduler queue if you are using the asynchronous functions supplied in the Solstice EM APIs.

Adding a callback to the scheduler queue involves:

- Initializing an instance of `Waiter` by calling an asynchronous function
- Registering the callback by calling the `when_resp` function on the `Waiter` instance that you initialized
- Calling the `send_resp` function on the `Waiter` instance that you initialized to add the registered callback to the scheduler queue

In the call to `send_resp`, specify a pointer to the data that the scheduler passes to the callback when the callback is run. Your callback must include code for converting the data into a `CurrentEvent` instance when the callback is run. If you

want to process the data, in your callback, call functions of the `CurrentEvent` class for processing the data. For more information, refer to Section 8.3.3.3 “Writing Code for Handling Responses From Managed Objects” on page 8-15.

8.4 Verifying and Changing the Status of an Asynchronous Operation

To verify or change the status of an asynchronous operation, use the instance of the `Waiter` class returned by the function you called to initiate the asynchronous operation. The instance of `Waiter` returned enables you verify or change the status of an asynchronous operation by:

- Verifying the result of an asynchronous operation
- Cancelling an asynchronous operation
- Specifying or extending the timeout of an operation

Note – The `start_create` function returns an instance of the `Result` class, not the `Waiter` class. Consequently, you cannot check or change the status of an asynchronous operation to add a managed object to the MIS.

8.4.1 Verifying the Result of an Asynchronous Operation

To enable your application to take different actions depending on whether an asynchronous operation succeeds, verify the result of the operation. For example, verify the result of an asynchronous operation to notify the user, or take some other recovery action, if the operation fails.

To verify the result of an asynchronous operation, call the `get_except` function of the `Waiter` class.

The `get_except` function returns one of the following:

- If the operation failed, `get_except` returns a pointer to an instance of the `ExceptionType` class. This instance provides information on why the operation failed.
- If the operation succeeded, `get_except` returns `NULL`.

The value returned by the `get_except` function is valid only while the operation is outstanding.

Note – The procedure for verifying the result of an asynchronous operations is different from that for asynchronous operations described in Section 4.1.2 “Using the `get_error_type` Function” on page 4-2. The `get_error_type` and `get_error_string` functions of the `Waiter` class return information about functions of the `Waiter` class, not about the operation represented by the instance of `Waiter`.

To obtain information on why an asynchronous operation failed, call functions of the `ExceptionType` class as follows:

- To obtain the exception type, call the `name` function of the `ExceptionType` class.
- To obtain the reason for the failure, call the `reason` function of the `ExceptionType` class.

Code for verifying the result of an asynchronous operation is shown in CODE EXAMPLE 8-11.

CODE EXAMPLE 8-11 Verifying the Result of an Asynchronous Operation

```
...
#include <pmi/hi.hh>                // High Level PMI
...
// WARNING: get_except() returns a 0 pointer if operation successful.
// Following code can also trigger segmentation faults if error occurs
// with SEM 3.0. Bottom line: AVOID USE OF ASYNC FUNCTIONALITY

if (cell_waiter.get_except()) {
    fprintf(stderr, "Waiter: Exception: %s, Reason: %s\n",
        (cell_waiter.get_except()->name(),
        (cell_waiter.get_except()->reason());
}
```

If the asynchronous operation represented by the instance of `Waiter` named `cell_waiter` fails, an error message is written to `stderr`. The text returned by the calls to the `name` and `reason` functions of `ExceptionType` is incorporated in the message written to `stderr`.

8.4.2 Cancelling an Asynchronous Operation

Cancel an asynchronous operation that you no longer require to be completed.

To cancel an outstanding asynchronous operation, call the `cancel` function of the `Waiter` class.

8.4.3 Changing the Timeout of an Asynchronous Operation

The timeout of an asynchronous operation specifies the maximum length of time an application allows for an asynchronous operation to finish. If an asynchronous operation is not completed within this length of time, the operation is cancelled.

Change the timeout of an asynchronous operation if you want the timeout to change in response to an event. For example, if your application is waiting for several events, change the timeout each time an event is received so that your application only times out if the interval between events exceeds some threshold.

To change the timeout of an asynchronous operation, call the `waitmore` function of the `Waiter` class. In the call to the `Waitmore` function, you have to specify the timeout. The timeout is a `typedef double` that represents the length of the timeout in seconds.

Code for changing the timeout of an asynchronous operation is given in CODE EXAMPLE 8-12.

CODE EXAMPLE 8-12 Changing the Timeout of an Asynchronous Operation

```
...
#include <pmi/hi.hh>           // High Level PMI
...
// Nonblocking Example (note: boot_cb not shown in this example):
    Waiter waiter1 = cell_image.start_create(Callback(boot_cb, 0));
    waiter1.waitmore(3.0); // Specify 3 sec. timeout for creation operation
...
```

In this example, the timeout for an asynchronous operation to add a managed object to an MIS is changed to 3.0 seconds.

Encoding and Decoding Complex ASN.1 Values

In the Solstice EM environment, attribute values in management requests, responses and event notifications are represented in a machine-independent format for transmission over a network. The format used is defined in ITU-T X.208/ISO-8824 *Specification of Abstract Syntax Notation One (ASN.1)*. This standard defines several complex data types and enables you to define your own custom data types. When an application sends a request to set an attribute value represented by a complex data type, the application must encode this value for transmission over a network. When an application receives an attribute value represented by a complex data type (for example in a response or an event notification) the application must decode this value to extract the information the value contains.

This chapter explains how to encode and decode complex ASN.1 values.

- Section 9.1 “Introduction to the Morf Class” on page 9-1
- Section 9.2 “Creating Complex ASN.1 Values” on page 9-2
- Section 9.3 “Parsing Complex ASN.1 Values” on page 9-9
- Section 9.4 “Decoding Complex ASN.1 Values” on page 9-24
- Section 9.5 “Using the MorfBuilder Class” on page 9-32

9.1 Introduction to the Morf Class

The high-level Portable Management Interface (PMI) API provides the `Morf` (mysterious object related to framework) class for representing complex ASN.1 values. To simplify the encoding and decoding of complex ASN.1 values, use the `Morf` class for:

- Creating complex ASN.1 values
- Parsing complex ASN.1 values to get information on their structure and content
- Decoding the values stored in complex ASN.1 values

Every instance of the `Morf` class is associated with an instance of the `Syntax` class. The `Syntax` class represents an ASN.1 type loaded into the metadata repository (MDR). The `Morf` class provides a simple way to transform string data into `Asn1Value` data in the underlying `Syntax` instance or to decode `Asn1Value` data into string data. Constructing and decoding complex data values by using strings is often much simpler than using the `Asn1Value` class directly.

`Morf` instances that represent complex data values can be used to pass data as a single instance to other functions and applications. This can be particularly useful if the data type of the value is unknown until run time, such as with a `CHOICE` type. In many cases where an application requires an `Asn1Value` instance, it is easier to construct a `Morf` instance and then use its `get_value` function to create the `Asn1Value` instance.

The `MorfBuilder` class provides additional facilities for building `Morf` instances that represent `CHOICE`, `SET`, `SET OF`, `SEQUENCE`, and `SEQUENCE OF` data types. The `MorfBuilder` class relies on constructing or decoding a value as a collection of `Morf` instances, so you need to understand how to work with the `Morf` class. Working with the `MorfBuilder` class is explained in Section 9.5 “Using the `MorfBuilder` Class” on page 9-32.

9.2 Creating Complex ASN.1 Values

To enable an application to send a request to set a complex ASN.1 value, you must create the value. To create a complex ASN.1 value, create and initialize an instance of the `Morf` class from any of the following:

- Data represented as strings
- Values that are instances of primitive types
- Other `Morf` instances

9.2.1 Creating a `Morf` Instance From String Data

You can represent the values in a complex value as a string. You can use this string directly in the `Morf` constructor or by calling the `set` function of the `Morf` class. The `Morf` class takes care of converting the string into the underlying `Asn1Value` instances contained in the complex value.

Most complex values contain lists of other values. Any value defined as a `SET`, `SET OF`, `SEQUENCE`, or `SEQUENCE OF` is considered a list. Lists can contain scalar values as well as other lists. Scalar values are represented literally as strings such as `"32"`, `"4.104"`, or `"Satellite A"`. Lists require additional syntax to mark the start and end of the list and to separate the list members.

9.2.1.1 Representing Complex Values as Strings

To represent a list as a string, enclose all of the list data in braces, { and }, and separate members with a comma.

Use the name of an `ENUMERATED` type in the string.

Consider the syntax of the `DestructSet` type in the satellite example, as shown in CODE EXAMPLE 9-1.

CODE EXAMPLE 9-1 ASN.1 Syntax of `DestructSet`

```
Checksum ::= SET OF OBJECT IDENTIFIER

DestructSet ::= SET OF SEQUENCE {
    name      GraphicString,
    value      Integer32,
    checksum   CheckSum
}
```

An instance of the `DestructSet` type is a set the members of which are sequences, and each sequence contains a set (`checksum`). An instance of the `DestructSet` type that contains two members is represented as a string as follows:

```
"{{name \"Code 1\", value 6753, checksum { 1 34 12 }}, {name \"Code
B\", value 9345, checksum { 3 12 5 }}}
```

9.2.1.2 Constructing a `Morf` Instance From a String

When you construct a `Morf` instance, you probably get the data for the `Morf` instance's attributes from various sources. Convert the data to `DataUnit` strings, then add the braces and commas required to order the data in sets and sequences. You can only build a `Morf` instance from a string that has values for all of the attributes in the underlying `Syntax` instance. You cannot update only a subset of the `Morf` instance's attributes.

If you need to build the subcomponents of a `Morf` instance at different times, consider building a `Morf` instance for each subcomponent. You can create an array of `Morf` instances from the subcomponents, and build a complex `Morf` instance that contains those subcomponents from the array. See Section 9.2.5 “Creating Complex `Morf` Instances From Other `Morf` Instances” on page 9-8.

CODE EXAMPLE 9-2 shows how to construct a `destructSet` from string data. This code creates a `Syntax` instance that represents the `destructSet` type, builds a string representation of a data set, then shows two ways of creating a `Morf` instance.

CODE EXAMPLE 9-2 Constructing a Morf From a String

```
...
#include <pmi/hi.hh>
#include <rw/cstring.h>
...
// CheckSum ::= SET OF OBJECT IDENTIFIER

// DestructSet ::= SET OF SEQUENCE {
//     name      GraphicString,
//     value     Integer32,
//     checkSum  CheckSum
// }

// Assume em_mis is an existing Platform instance
// previously connected to the MIS

Syntax syn(DU("destructSet"), em_mis);
RWCString destrSet, destrSequence1, destrSequence2;

//Build the string representation of the set
destrSequence1 =
    "{ name \"Code A\", value 6753, checkSum { 1 34 12 }}";
destrSequence2 =
    "{ name \"Code B\", value 9345, checkSum { 3 12 5 }}";
destrSet = "{" + destrSequence1 + ","
    + destrSequence2 + "}";

//Construct a Morf using the string and Syntax syn
Morf m1(syn, destrSet.data());

//Construct an empty Morf, then add the data
Morf m2(syn);
m2 = m2.set(destrSet);
if (m2.get_error_type != PMI_SUCCESS) {
    //handle the error
}
...
```

9.2.2 Creating Simple Morf Instances

The `set_str` function converts a string to an arbitrary ASN.1 *scalar* value, such as `graphicString`, `topoBoolean`, or `INTEGER`. The `set_str` function does not work on values that are instances of list types such as `SET` or `SEQUENCE`. Use `set` for values that are instances of list types.

It may be easier to use `assign` numeric data directly to a scalar, rather than first converting it to a character string. There are several functions for using numeric data to set the value of scalars based on numeric types.

The numeric data functions work on `Morf` instances that represent a scalar value the base type of which is one of the following:

- `BOOLEAN`
- `ENUMERATED`
- `INTEGER`
- `OCTET STRING`
- `REAL`

The functions for setting scalar values all return a new `Morf` instance. The functions in TABLE 9-1 set the ASN.1 value of scalar-valued `Morf` instances.

TABLE 9-1 Functions for Assigning Scalar Values to a `Morf` Instance

Function	Description
<code>set_dbl</code>	Use a variable of type <code>double</code> to assign a value to a scalar-valued numeric <code>Morf</code> instance.
<code>set_gint</code>	Use a reference to a variable of type <code>GenInt</code> to assign a value to a scalar-valued numeric <code>Morf</code> instance. Most types derived from <code>int</code> (<code>long</code> , <code>I32</code> , <code>U32</code>) can be cast as <code>GenInt</code> .
<code>set_long</code>	Use a variable of type <code>long</code> to assign a value to a scalar-valued numeric <code>Morf</code> instance.
<code>set_str</code>	Use a string to assign a value to a scalar-valued <code>Morf</code> instance of any ASN.1 type (for example: <code>GraphicString</code> , <code>Integer32</code> , or <code>BOOLEAN</code>). In the call to <code>set_str</code> , you also have to specify format bits to control the format of the string. For more information, see Section 9.4.1.2 “Controlling the String Representation of a <code>Morf</code> Instance” on page 9-26.

9.2.3 Selecting the Type for a CHOICE Value

Before you assign a value to a CHOICE type, call the `set_memname` function to select which real syntax this instance of the type should use.

A `Morf` instance that corresponds to a CHOICE value has no definite syntax until one of the types in the CHOICE is selected. If you attempt to assign a value to an uninitialized `Morf` instance of type CHOICE, the `Morf` instance will not understand which syntax to use.

CODE EXAMPLE 9-3 shows how to use `set_memname`.

CODE EXAMPLE 9-3 Selecting the Type for a CHOICE Value

```
...
#include <pmi/hi.hh>
...
// Assume Syntax syn is associated with a GeoLocation:
//
//   GeoLocation ::= CHOICE {
//       null    NULL,
//       value   SEQUENCE {
//           longitude REAL,
//           latitude REAL
//       }
//   }
//
...
Morf geoLocation(syn);
geoLocation.set_memname("value"); // use value syntax, not null
geoLocation.set("{longitude 122.35, latitude 38.35}");
...
```

9.2.4 Creating a Morf Instance for ASN.1 ANY Values

Call the `set_any` function to assign another Morf instance associated with the real syntax of the value to a Morf instance associated with an ANY or ANY DEFINED BY syntax. In the call to `set_any`, specify the Morf instance associated with the real syntax of the value.

ANY and ANY DEFINED BY values are harder to build programmatically because the actual ASN.1 type, syntax, and valid values for the data are context sensitive. For example, the `AttributeValueAssertion` type has the following syntax:

CODE EXAMPLE 9-4 ASN.1 Syntax of `AttributeValueAssertion`

```
AttributeType ::= OBJECT IDENTIFIER
AttributeValue ::= ANY

AttributeValueAssertion ::= SEQUENCE {
    type AttributeType,
    value AttributeValue
}
```

In this type, the kind of data referenced by the type field determines the syntax of the value field.

Assigning a value to a Morf instance of type ANY involves:

- Creating a Morf instance (m1) based on the Syntax instance of type ANY
- Creating a second Morf instance (m2) by using the real Syntax instance of the value to be assigned
- Assigning a value to the second Morf instance by calling, for example, `set` or `set_str`
- Calling `set_any` on m1, specifying m2 as follows:

```
m1.set_any(Morf m2(realSyntax, DU(realData)));
```

CODE EXAMPLE 9-5 shows the construction of an `AttributeValueAssertion` ASN.1 value. The value represents the attribute value assertion: `topoBoolean = TRUE`.

CODE EXAMPLE 9-5 Assigning a Value to an Instance of the ASN.1 ANY Type

```
...
#include <pmi/hi.hh>
...
```

CODE EXAMPLE 9-5 Assigning a Value to an Instance of the ASN.1 ANY Type *(Continued)*

```
// Create Syntax for AttributeValueAssertion, AttributeType
// AttributeValue and TopoBoolean (real type of AttributeValue
// Assume em_mis is an existing Platform instance
// previously connected to the MIS
Syntax avaSyn("attributeValueAssertion", em_mis);
Syntax attSyn("attributeType", em_mis);
Syntax atvSyn("attributeValue", em_mis);
Syntax tbSyn("topoBoolean", em_mis);

// get the OID of TopoBoolean to create a Morf
Oid tbOid;
Result r;
if ((r = OidNameRegistry::find_oid_by_name("topoBoolean", tbOID))
    != OK ) {
    // Handle error
}
Morf attMf(attSyn, tbOid.data());

// create an AttributeValue Morf (type ANY) and assign a
// Morf of type TopoBoolean to it
Morf atvMf(atvSyn);
Morf tbMf(tbSyn, "TRUE");
atvMf = atvMf.set_any(tbMf);

// Assemble the type and the value into one Morf
Array(Morf) ava(2);

ava[0] = attMf;
ava[1] = atvMf;

Morf avaMf(avaSyn, ava);
...
```

9.2.5 Creating Complex Morf Instances From Other Morf Instances

CODE EXAMPLE 9-5 shows how to build a Morf instance by first building Morf instances that represent its component values. For SEQUENCE, SEQUENCE OF, SET, and SET OF type Morf instances, you can create an array that has the list members as elements, then use that array to construct a new Morf instance.

You can also use the Queue class to construct a Morf instance from a queue (that is, an ordered list) of Morf instances.

The syntax of the constructor for creating a `Morf` instance from an array of `Morf` instances is as follows:

```
Morf(Syntax& syn, Array(Morf)& ma)
```

The syntax of the constructor for creating a `Morf` instance from a queue of `Morf` instances is as follows:

```
Morf(Syntax& syn, class Queue(MorfElem)& mq)
```

These constructors are useful for simple `SET` and `SEQUENCE` values. More complex values may involve `CHOICE` values, `SET` values within `SET` values, or `SEQUENCE OF SET` values, for example. For constructing any moderately complex `Morf` instance, it is usually easier to use the `MorfBuilder` class. See Section 9.5 “Using the `MorfBuilder` Class” on page 9-32 for information on using the `MorfBuilder` class.

Note – A `Morf` instance does not change after you have constructed it. Changes made to an array or a queue after the `Morf` instance is constructed are not reflected in the `Morf` instance.

9.3 Parsing Complex ASN.1 Values

You usually derive a complex ASN.1 value from the Solstice EM platform as an instance of the `Morf` class without knowing its explicit type and structure. Parsing the `Morf` instance that represents a complex ASN.1 value enables you to understand the structure of the instance and its ASN.1 syntax. When you know the type and structure, you can extract the data you need from the `Morf` instance or run different code based on the type of data received.

Reading or modifying values in a `Morf` instance involves:

- Finding the member of the instance that you want (for example, a particular member of a `SET` or an attribute in a `SEQUENCE`)
- Verifying the ASN.1 type of the selected member
- Understanding any constraints on values for the selected type (valid ranges or size constraints)

You need to parse a `Morf` instance when the data type of a value is unknown, but also sometimes when the type is known. For `CHOICE` types, you must parse the `Morf` instance to determine which of the possible types the data represents.

The `Morf` class provides functions to discover the structure of a `Morf` instance and all of the data types of all the values it contains.

9.3.1 Structure of `Morf` Instances

Every `Morf` instance is associated with a `Syntax` instance that represents an ASN.1 type. Every ASN.1 type is either a scalar type or a constructed type. A constructed type is either a *list type* (namely: `SET`, `SET OF`, `SEQUENCE`, or `SEQUENCE OF`) or a `CHOICE` type.

The `ANY` or `ANY DEFINED BY` is an open type. The actual syntax of the data contained in an instance of the type could be any valid ASN.1 type. For `ANY DEFINED BY`, the possible syntax choices are restricted to types allowed by the type named in the declaration.

Lists and scalars may be combined with other lists or scalars to create more complex types. For example, a `SEQUENCE` can contain scalars and a `SET` of instances of the type `SEQUENCE`.

Functions for parsing a `Morf` instance enable you to decompose a `Morf` instance into the values it contains, decompose those values, and so on, until you have identified all of the fundamental scalar values and how they are contained in other values.

Any type or a corresponding `Morf` instance can be only one of the following types:

- `ANY`
- `CHOICE`
- List type
- Scalar

Parsing a `Morf` instance involves:

- Determining the type associated with the `Morf` instance
- Doing one of the following, depending on the type:
 - If the type is `CHOICE` or `ANY`, extracting the actual syntax of the data as a new `Morf` instance then resuming parsing
 - If the type is a list type, breaking the list into an array or queue of `Morf` instances and resuming parsing each array or queue

9.3.2 Overview of Functions for Parsing Morf Instances

The functions of the `Morf` class for obtaining information about the structure of a `Morf` instance are listed in TABLE 9-2.

TABLE 9-2 Functions for Parsing `Morf` Instances

Morf Class Function	Description
<code>extract</code>	Extracts the specified element as a <code>Morf</code> instance. Use a navigation string to specify an attribute name or element number in the current <code>Morf</code> instance or one of its contained values. Separate contained values with a dot (.). For example, the navigation string <code>GeoLocation.value.latitude</code> extracts the value of <code>latitude</code> from the value element in the attribute <code>GeoLocation</code> . The definition of the <code>GeoLocation</code> type is shown in CODE EXAMPLE 9-3.
<code>get_member_names</code>	Returns an array of the <code>Morf</code> instance's attribute names. If the <code>Morf</code> instance represents an instance of a <code>CHOICE</code> type, returns the attribute names of the type chosen for the instance.
<code>get_memname</code>	Returns the attribute name of the <code>ASN.1</code> type associated with the <code>Morf</code> instance. If the <code>Morf</code> instance represents an instance of a <code>CHOICE</code> type, returns the attribute names of the type chosen for the instance.
<code>get_platform</code>	Returns the <code>Platform</code> instance associated with the <code>Morf</code> instances's <code>Syntax</code> instance.
<code>get_syntax</code>	Returns a <code>Syntax</code> instance that represents the syntax associated with the <code>Morf</code> instance.
<code>get_type</code>	Returns an <code>Asn1Type</code> instance that corresponds to the <code>Morf</code> instance's <code>ASN.1</code> type.
<code>has_value</code>	Use only to check if the <code>Morf</code> instance has a value assigned to it. If there is no value assigned or if the value is <code>NULL</code> , this function returns a null pointer.
<code>is_any</code>	Returns <code>TRUE</code> if the <code>Morf</code> instance represents an instance of type <code>ANY</code> or <code>ANY DEFINED BY</code> .
<code>is_choice</code>	Returns <code>TRUE</code> if the <code>Morf</code> instance represents an instance of type <code>CHOICE</code> .
<code>is_list</code>	Returns <code>TRUE</code> if the <code>Morf</code> instance represents an instance of type <code>SEQUENCE</code> , <code>SEQUENCE OF</code> , <code>SET</code> , or <code>SET OF</code> .
<code>is_sequence</code>	Returns <code>TRUE</code> if the <code>Morf</code> instance represents an instance of type <code>SEQUENCE</code> or <code>SEQUENCE OF</code> .
<code>is_set</code>	Returns <code>TRUE</code> if the <code>Morf</code> instance represents an instance of type <code>SET</code> or <code>SET OF</code> .
<code>num_elements</code>	Returns the number of elements in a set or sequence. Valid only on list type <code>Morf</code> instances.
<code>split_array</code>	Returns the elements in a value that is an instance of a list type as an array of <code>Morf</code> instances. Valid only on list type <code>Morf</code> instances.
<code>split_queue</code>	Returns the elements in a value that is an instance of a list type as a queue of <code>MorfElem</code> types. Valid only on list type <code>Morf</code> instances.

TABLE 9-3 lists functions provided by the `Asn1Type` class that are also useful for parsing `Morf` instances. Call the `get_type` function of the `Morf` class to extract an `Asn1Type` instance from a `Morf` instance.

TABLE 9-3 Functions of the `Asn1Type` Class For Parsing `Morf` Instances

<code>Asn1Type</code> Class Function	Description
<code>get_bit_string_identifiers</code>	Valid for <code>BIT STRING</code> types. Returns an array that contains the name and position of possible values defined for the string.
<code>get_enum_identifiers</code>	Valid for <code>ENUMERATED</code> types. Returns an array that contains the string identifier and numeric value for all values defined in the enumeration.
<code>get_range</code>	Valid for types derived from <code>INTEGER</code> or <code>REAL</code> . Returns the lowest and highest possible value for this type. If a range cannot be determined for this type, <code>get_range</code> returns <code>NOT_OK</code> .
<code>get_size_constraint</code>	Valid for types derived from <code>BIT STRING</code> , <code>OCTET STRING</code> , <code>SEQUENCE OF</code> , and <code>SET OF</code> . Returns the smallest and largest possible size for this type. If there are no size constraints for this type, <code>get_size_constraint</code> returns <code>NOT_OK</code> .

9.3.3 Parsing CHOICE Values

Use `is_choice` to determine if a `Morf` instance represents a `CHOICE` type.

If you only want to know what type is chosen, use `get_memname`. For `CHOICE` types, `get_memname` returns the attribute name of the chosen type.

If you want to parse the instance further, call `extract` to extract a `Morf` instance associated with the chosen syntax. In the call to `extract`, specify a navigation string by calling `get_memname` to identify the name of the chosen instance.

CODE EXAMPLE 9-6 shows how to get information about a `CHOICE` value by using `get_memname` and `extract`.

CODE EXAMPLE 9-6 Extracting Data From a `CHOICE` Value

```
...
#include <pmi/hi.hh>
...
// Assume we have a CHOICE Morf in morf
if (morf.is_choice()) {
    cout << "CHOICE attribute selected is : ";
    cout << morf.get_memname();
}
```

CODE EXAMPLE 9-6 Extracting Data From a CHOICE Value *(Continued)*

```
// Now replace morf with the chosen Syntax for further
// parsing.
Morf tmp = morf;
morf = tmp.extract(tmp.get_memname());
if (!morf) {
    // handle the error
}

// Continue parsing with morf
...
```

9.3.4 Parsing List Values

Parsing a list value involves:

- Determining if the `Morf` instance represents a list
- Determining how many members are in the list
- Splitting the list into an array or queue of new `Morf` instances
- Getting the type of members of the list

The sample code in Section 9.3.7 “Example of Parsing a Morf Instance” on page 9-23 shows how to parse list values.

9.3.4.1 Determining That a Morf Instance Represents a List

The following functions enable you to determine if a `Morf` instance represents a list and, if so, whether the list is a sequence or a set:

- `is_list` returns `TRUE` for any list.
- `is_sequence` returns `TRUE` for any instance of type `SEQUENCE` or `SEQUENCE OF`.
- `is_set` returns `TRUE` for any instance of type `SET` or `SET OF`.

9.3.4.2 Getting the Number of Members in a List

If a `Morf` instance represents a list, call `num_elements` to determine how many members are in the list.

9.3.4.3

Splitting a List Into an Array or Queue of New Morf Instances

If you need to parse the members of a list, you first need to extract the members into new Morf instances. The following functions return each element in a list as a new Morf instance:

- `split_array` returns an array of Morf instances.
- `split_queue` returns a queue of Morf instances.

You then need to parse each element in the array or queue.

CODE EXAMPLE 9-7 shows how to use a queue to determine whether a list contains a particular value.

CODE EXAMPLE 9-7 Using a Queue to Parse a List

```
...
#include <pmi/hi.hh>
...
// SatelliteSeq ::= SEQUENCE {
//   name      GraphicString,
//   value      Integer32,
//   checksum   CheckSum
// }

SatelliteData ::= SET OF SatelliteSeq

// Assume Morf morf is of type SatelliteData

// Create the value we are looking for
// Assume em_mis is an existing Platform instance
// previously connected to the MIS
Syntax syn("Integer32", em_mis);
Morf testMf(syn, "7777");

// Iterate over the SatelliteData looking for a SatelliteSeq
// that matches our test value
if (morf.is_set() && (morf.num_elements() > 0) ) {
    Queue(MorfElem) setQ = morf.split_queue();
    MorfElem mfe;
    for (mfe = setQ.fiq(); mfe; mfe = setQ.niq(mfe)) {
        // Each queue item is a SatelliteSeq; check its 'value'
        Morf testVal = mfe.mf->extract("value");
        if (!testVal) continue;
        if ( testVal == testMf ) {
            // Found, return this SatelliteSeq
            return Morf(mfe.mf);
        }
    }
}
```

CODE EXAMPLE 9-7 Using a Queue to Parse a List (Continued)

```
        }
    }
}
return Morf();    // Not found or empty set
...
```

9.3.4.4 Getting the Types of Members of a List

You may need only the types of the members of a list. For example, if the type of a member is `CHOICE` or `ANY`, you can only know at runtime the actual type of the member. To get the actual type of a member that is an instance of the `CHOICE` or `ANY` type, call the `get_member_names` function.

The `get_member_names` function returns an array the ASN.1 of types in of the members of list.

To examine the array, call functions of the `DataUnit` class to get information about the data types.

9.3.5 Getting Objects Associated With a `Morf` Instance

An instance of each of the following classes is associated with a `Morf` instance:

- `Platform`
- `Syntax`
- `Asn1Type`

To retrieve an instance, call one of the functions of the `Morf` class listed in TABLE 9-4.

TABLE 9-4 Functions for Retrieving Information About the Type Instance

Class	Function	Returns
<code>Platform</code>	<code>get_platform</code>	A reference to the associated <code>Platform</code> instance
<code>Syntax</code>	<code>get_syntax</code>	The instance of the <code>Syntax</code> class associated with the <code>Morf</code> instance's syntax
<code>Asn1Type</code>	<code>get_type</code>	The <code>Asn1Type</code> instance that represents the <code>Morf</code> instance's underlying ASN.1 type

9.3.6 Getting Metainformation About the ASN.1 Type of a Morf Instance

To test if a value is assigned to a Morf instance, call the `has_value` function on the Morf instance. Use the `has_value` function only to test if a value is assigned to a Morf instance. Do not attempt to use pointer that the `has_value` function returns.

The `Asn1Type` class provides functions for getting:

- Identifiers for a BIT STRING value
- Identifiers for an ENUMERATED value
- The range of a value of a type or subtype of REAL or INTEGER
- The size constraints of a value of a type or subtype of BIT STRING, OCTET STRING, SEQUENCE OF, or SET OF

Use the `get_type` function of the Morf class to get an `Asn1Type` instance that corresponds to the Morf instance. Use the returned `Asn1Type` instance to call functions for getting metainformation about values. For example:

```
Result r = morf.get_type().get_enum_identifiers(idArray);
```

9.3.6.1 Getting Identifiers for a BIT STRING Value

To get identifiers for a BIT STRING value, call the `get_bit_string_identifiers` function of the `Asn1Type` class.

The syntax of `get_bit_string_identifiers` is as follows:

```
Result get_bit_string_identifiers(Array(Asn1NamedNumber) &idents)
```

If this function returns OK, the *idents* array holds the identifiers and associated positions for the members in an ASN.1 BIT STRING value.

CODE EXAMPLE 9-8 shows how to use `get_bit_string_identifiers`.

9.3.6.2 Getting Identifiers for an ENUMERATED Value

To get identifiers for an ENUMERATED value, call the `get_enum_identifiers` function of the `Asn1Type` class.

The syntax of `get_enum_identifiers` is as follows:

```
Result get_enum_identifiers(Array(Asn1NamedNumber) &idents)
```

If this function returns OK, the *idents* array holds the identifiers and associated values for the values defined in an ASN.1 ENUMERATED value.

CODE EXAMPLE 9-8 shows how to use `get_bit_string_identifiers` and `get_enum_identifiers` to print the possible values for BIT STRING and ENUMERATED types:

CODE EXAMPLE 9-8 Obtaining BIT STRING and ENUMERATED Identifiers

```
...
#define BIT_STRING 1
#define ENUM      2
#include <pmi/hi.hh>
...

void show_ids(Array(Asn1NamedNumber) &idents, int type) {
    Asn1TypeInt    int_type(AK_INTEGER);
    DU             ident;
    GenInt         numbvalue;
    U32            i;
    Asn1Value      aslnnumber;
    Asn1ParsedValue number;

    cout << "Number of identifiers: " << idents.size << endl;
    for (i=0; i<idents.size; i++) {
        number = idents[i].num;
        ident = idents[i].name;
        if (number) {
            aslnnumber = number.get_real_val(int_type);
            aslnnumber.decode_int(numbvalue);
        }

        cout << "Identifier # => " << i << " Name is => ";
        cout << ident.chp() << ";";
        cout << " Identifier ";
        cout << ((type == ENUM) ? "value" : "position") ;
        cout << " is => ";
        if (number) {
            cout << I32(numbvalue);
        } else {
            cout << "NULL" ;
        }
    }
}
```

CODE EXAMPLE 9-8 Obtaining BIT STRING and ENUMERATED Identifiers (Continued)

```
        cout << endl ;
    } // end for()
} // end show_idsents()

...
// bitStrMf is a Morf of type BIT STRING
// enumMf is a Morf of type ENUMERATED
...
Result r;
Array(Asn1NamedNumber) newidents;

// BIT STRING
r = bitStrMf.get_type().get_bit_string_identifiers(newidents);
if (r == NOT_OK) {
    cout << "Failed to get BIT STRING identifiers!" << endl;
} else {
    show_idsents(newidents, BIT_STRING);
}

// ENUMERATED
r = enumMf.get_type().get_enum_identifiers(newidents);
if (r == NOT_OK) {
    cout << "Failed to get ENUMERATED identifiers!" << endl;
} else {
    show_idsents(newidents, ENUM);
}
...
```

9.3.6.3 Getting the Range of a Value of a Type or Subtype of REAL or INTEGER

To get the range of a value of a type or subtype of REAL or INTEGER, call the `get_range` function of the `Asn1Type` class.

The syntax of `get_range` is as follows:

```
Result get_range(Asn1ParsedValue &lower,
                 Boolean          &lower_open,
                 Asn1ParsedValue &upper,
                 Boolean          &upper_open)
```

The `get_range` function returns `NOT_OK` if it is called on an instance that does not represent an ASN.1 type or subtype of `INTEGER` or `REAL`.

If the function returns OK, *lower* is set to the lowest possible value for the type and *upper* is set to the highest possible value.

ITU-T X.208/ISO-8824 *Specification of Abstract Syntax Notation One (ASN.1)* uses MIN and MAX to define the lower and upper ranges of subtypes of INTEGER and REAL. The PMI library encodes MIN and MAX as NULL *Asn1ParsedValue* values. Therefore, you must make sure *lower* and *upper* are not NULL before attempting to decode them.

If the function returns OK, the *lower_open* and *upper_open* boolean variables indicate whether the lower and upper range limits are open (TRUE). If a range limit is not open, the corresponding variable (*lower_open* or *upper_open*) is set to FALSE.

The code in CODE EXAMPLE 9-9 shows how to use *get_range* to parse and decode the range limits for REAL and INTEGER types.

CODE EXAMPLE 9-9 Obtaining the Range Limits for a Value

```
...
#include <pmi/hi.hh>
...
void show_range(Asn1Type rtype) {
    Asn1TypeInt      int_type(AK_INTEGER);
    Asn1Type         real_type(AK_REAL);
    Result           r;
    int              is_real = 0;
    Asn1Value        asnllower, asnlupper;
    Asn1ParsedValue  lower, upper;
    GenInt           low, up;
    Boolean          lower_open, upper_open;
    double           dbl;

    if (rtype.base_kind() == AK_REAL) {
        is_real = 1;
    } else if (rtype.base_kind() != AK_INTEGER) {
        // Not valid type for get_range()!
        cout << "Cannot determine range for type." << endl;
        return;
    }

    r = rtype.get_range(lower, lower_open, upper, upper_open);
    if (r == NOT_OK) {
        cout << "get_range() failed!" << endl;
        return;
    }

    cout << '\t';
    if (lower) {
```

CODE EXAMPLE 9-9 Obtaining the Range Limits for a Value *(Continued)*

```
    if (is_real) { //REAL
        asnlower = lower.get_real_val(real_type);
        asnlower.decode_real(dbl);
        cout << "Lower range is " << dbl << ".";
    } else {      // INTEGER
        asnlower = lower.get_real_val(int_type);
        asnlower.decode_int(low);
        cout << "Lower range is " << I32(low) << ".";
    }

    if (lower_open == TRUE) {
        cout << "Lower range is open." << endl;
    } else {
        cout << "Lower range is closed." << endl;
    }
} else { // lower is NULL, range is MIN
    cout << "Lower range is MIN." << endl;
}

cout << '\t';
if (upper) {
    if (is_real) { //REAL
        asnupper = upper.get_real_val(real_type);
        asnupper.decode_real(dbl);
        cout << "Upper range is " << dbl << ".";
    } else {      // INTEGER
        asnupper = upper.get_real_val(int_type);
        asnupper.decode_int(up);
        cout << "Lower range is " << I32(up) << ".";
    }

    if (upper_open == TRUE) {
        cout << "Upper range is open." << endl;
    } else {
        cout << "Upper range is closed." << endl;
    }
} else { // upper is NULL, range is MAX
    cout << "Upper range is MAX." << endl;
}

} // end show_range()

...
// Morf morf represents a type we think is
// derived from INTEGER or REAL
```

CODE EXAMPLE 9-9 Obtaining the Range Limits for a Value (*Continued*)

```
if (morf.get_type()) {
    if ((morf.get_type().base_kind() == AK_INTEGER) ||
        (morf.get_type().base_kind() == AK_REAL))
        show_range(morf.get_type());
} else {
    cout << "morf not initialized!" << endl;
}
...
```

9.3.6.4 Getting the Size Constraints of a Value

To get the size constraints of a value of a type or subtype of BIT STRING, OCTET STRING, SEQUENCE OF, or SET OF, call the `get_size_constraint` function of the `Asn1Type` class.

The syntax of `get_size_constraint` is as follows:

```
Result get_size_constraint(Asn1ParsedValue &lower,
                           Boolean &lower_open,
                           Asn1ParsedValue &upper,
                           Boolean &upper_open)
```

If the function returns OK, *lower* is set to the smallest possible size for the type and *upper* is set to the largest possible size.

ITU-T X.208/ISO-8824 *Specification of Abstract Syntax Notation One (ASN.1)* uses MIN and MAX to define the lower and upper size limits of subtypes of BIT STRING, OCTET STRING, SEQUENCE OF, and SET OF. The PMI library encodes MIN and MAX as NULL `Asn1ParsedValue` values. Therefore, you must make sure *lower* and *upper* are not NULL before attempting to decode them.

If the function returns OK, the *lower_open* and *upper_open* boolean variables indicate whether the lower and upper size limits are open (TRUE). If a size limit is not open, the corresponding variable (*lower_open* or *upper_open*) will be set to FALSE.

The `get_size_constraint` function returns NOT_OK if it is called on an instance that does not represent an ASN.1 type or subtype of BIT STRING, OCTET STRING, SEQUENCE OF, or SET OF.

The code in CODE EXAMPLE 9-10 shows how to use `get_size_constraint` to parse and decode the size limits for BIT STRING, OCTET STRING, SEQUENCE OF, and SET OF types.

CODE EXAMPLE 9-10 Obtaining the Size Constraints of a Value

```
...
#include <pmi/hi.hh>
...
void show_size_constraint(Asn1Type stype) {
    char          buf[4096], *bufp;
    U32           i, buflen = 4096;
    Asn1TypeInt    int_type(AK_INTEGER);
    Result        r;
    Asn1Value      asnllower, asnlupper;
    Asn1ParsedValue lower, upper;
    GenInt         low, up;
    Boolean        lower_open, upper_open;

    r = stype.get_size_constraint(lower, lower_open, upper,
    upper_open);
    if (r == NOT_OK) {
        cout << "Could not get size constraints." << endl;
        return;
    }

    if (lower) {
        asnllower = lower.get_real_val(int_type);
        asnllower.decode_int(low);

        buflen = 4096;
        bufp = buf;
        int_type.format_value(asnllower, bufp, buflen, 0,
            TAG_EXPLICIT, DataUnit(), 0);
        cout << "Lower limit is " << buf << "." << endl;

        cout << "Lower limit is ";
        cout << ((lower_open == TRUE) ? "open" : "closed") ;
        cout << "." << endl;

    } else { // lower is NULL, constraint is MIN
        cout << "Lower limit is MIN." << endl;
    }

    if (upper) {
        asnlupper = upper.get_real_val(int_type);
        asnlupper.decode_int(up);
    }
}
```

CODE EXAMPLE 9-10 Obtaining the Size Constraints of a Value *(Continued)*

```
        buflen = 4096;
        bufp = buf;
        int_type.format_value(asnupper, bufp, buflen, 0,
                               TAG_EXPLICIT, DataUnit(), 0);
        cout << "Upper limit is " << buf << "." << endl;

        cout << "Upper limit is ";
        cout << ((upper_open == TRUE) ? "open" : "closed") ;
        cout << "." << endl;

    } else { // upper is NULL, constraint is MAX
        cout << "Upper limit is MAX." << endl;
    }

} // end show_size_constraint()

...
// Morf morf has been derived from the platform somehow...
if (morf.get_type()) {
    show_size_constraint(morf.get_type());
} else {
    cout << "morf not initialized!" << endl;
}
...
```

9.3.7 Example of Parsing a Morf Instance

The function `morf_split` in CODE EXAMPLE 9-11 shows a typical recursive algorithm for parsing any type of Morf instance. The example prints out type names and values for scalars, but you can use the same algorithm to do other things with the data.

CODE EXAMPLE 9-11 Sample Function for Parsing a Morf Instance

```
...
#include <pmi/hi.hh>
...
void
morf_split( Morf& m)
{
    // Check if the morf contains CHOICE,
    //
```

CODE EXAMPLE 9-11 Sample Function for Parsing a Morf Instance (*Continued*)

```
// If the morf contains CHOICE,
// extract the morf, then call the function recursively.
// If the morf does not contains CHOICE,
// Check if the morf contains a compound data value.
//
// If the morf contains a compound data value,
//     split the morf, then call the function recursively.
//
// If the morf contains scalar data value,
//     print the scalar data value.

if (m.is_choice()){
    morf_split(m.extract(DU()));
} else if (m.is_list()) {
    Array(Morf) mm = m.split_array();
    for (int i=0; i<mm.size; i++) {
        cout << "morf[";
        cout << i ;
        cout << "] = ";
        cout << mm[i].get_str().chp();
        cout << endl;
        morf_split(mm[i]);
    }
} else {
    cout << endl;
    cout << "---scalar value--->";
    cout << m.get_str().chp();
    cout << endl << endl << endl;
}
}
```

9.4 Decoding Complex ASN.1 Values

After you understand the structure of any Morf instance, you may want to extract values assigned to attributes in the instance. The Morf class provides functions for extracting or decoding data in a Morf instance. You can decode any Morf instance that has been initialized with data by:

- Getting a string representation of the Morf instance
- Extracting a value in the Morf instance as a new Morf instance
- Getting the value assigned to the Morf instance as an Asn1Value instance
- Getting scalar values assigned to a Morf instance

9.4.1 Getting a String Representation of a Morf Instance

To retrieve a string representation of a Morf instance's values, call the `get` function on the Morf instance. The `get` function works on any Morf instance. The `get` function returns a string that represents the Morf instance's structure and the values assigned to attributes.

9.4.1.1 Getting the Default String Representation of a Morf Instance

To get the default string representation of a Morf instance, call the `get` function on the Morf instance specifying 0 for the format bits parameter. The default representation of each value depends on the type of the value as shown in TABLE 9-5.

TABLE 9-5 Default String Representation of Values by Type in a Morf Instance

Type	Representation
SET SEQUENCE	The members of the SET or SEQUENCE value are enclosed in braces and separated by commas.
CHOICE ANY	The actual type represented by a CHOICE or ANY value is indicated by a label of the form " <i>type</i> :" in the string before the values. By default, the string contains identifiers of named numbers, such as ENUMERATED values.
BIT STRING	When they appear as numbers, BIT STRING values are enclosed in single quotes followed by B. For example, '01101101'B.
OCTET STRING	When they appear as hexadecimal values, OCTET STRING values are enclosed in single quotes followed by H. For example, '323A1F0A'H.
BOOLEAN	Boolean values are translated into the strings TRUE and FALSE.

CODE EXAMPLE 9-12 shows the ASN.1 type definition of the GeoLocation type.

CODE EXAMPLE 9-12 ASN.1 Type Definition of the GeoLocation Type

```
GeoLocation ::= CHOICE {  
    null NULL,  
    value SEQUENCE {  
        latitude REAL,  
        longitude REAL  
    }  
}
```

CODE EXAMPLE 9-13 shows the default string representation of a `GeoLocation` value returned by a call to `get`.

CODE EXAMPLE 9-13 Default String Representation of a `GeoLocation` Value

```
"value : {  
    122.35,  
    38.37 }"
```

9.4.1.2 Controlling the String Representation of a `Morf` Instance

You often extract a string representation of a `Morf` instance or one of its members by using the `get` function. Strings can represent any arbitrary data in a way that makes it easy to parse or to reuse the data to build new `Morf` instances. If you require a string representation that uses a different format than the default, specify the `format bits` parameter in calls to functions that use strings.

The `format bits` argument is a set of bit flags that specify how to format data in the output string. The default value is 0, and that is usually what you use.

Because the `Morf` class handles complex data as familiar string types, you sometimes want control over how to format the string. The `format bits` argument is used in `get` and `get_str`. The `set` and `set_str` functions also accept `format bits`, but only the `USE_EXPLICIT_CHOICE` flag is meaningful when a value is set.

To use the `format bits`, combine all of the flags you want to use using a bitwise OR:

```
cout << "mrf = " << mrf.get(USE_NUMERIC_NAMES|USE_C_ESCAPES);
```


TABLE 9-6 lists the identifiers for specifying format bits.

TABLE 9-6 Identifiers for Format Bits Arguments

Format Bit Identifier	Description
USE_NUMERIC_NAMES	Returns only the numbers of an object identifier. Do not attempt to translate OID components into names.
OMIT_NEWLINES	Removes newline characters from a string before it is returned.
USE_C_ESCAPES	Returns OCTET STRING values with special characters escaped by using C shell escape characters. Nonprinting characters are represented by \0, \a, \b, \f, \n, \t, \r, \v, or \nnn where <i>nnn</i> is the hexadecimal value of the character.
USE_EXPLICIT_TYPES	Prefixes values of type ANY or ANY DEFINED BY with a label that indicates the actual type of the value returned.
OMIT_SPACES	Strips spaces from a string before it is returned.
USE_HEX	Shows the hexadecimal value of each octet in OCTET STRING data.
USE_EXPLICIT_CHOICE	Prefixes values of type CHOICE with a label that indicates the actual type of the value returned.

9.4.2 Extracting a Value in a Morf Instance as a New Morf Instance

To extract a value in a Morf instance as a new Morf Instance, call the `extract` function on the existing Morf instance. The `extract` function retrieves a member of a Morf instance that represents a list.

Use the `extract` function when you want to work with a subcomponent of a complex Morf instance, especially when you want to extract one element from a SET or SEQUENCE. Unlike `split_array` and `split_queue`, `extract` enables you to name and extract a single item from a list.

The `extract` function is also useful for working with CHOICE types. The CHOICE syntax tells you nothing about the structure of the data. To get information on the structure of the data, use `extract` to extract a Morf instance that represents the syntax of the actual values assigned to the Morf instance.

In the call to `extract`, specify a navigation string. The navigation string indicates which member of a list Morf instance to extract. The navigation string is composed of one or more identifiers separated by periods (.). Each identifier can be one of the following:

- The name of an attribute in a `SEQUENCE`
- The name of an attribute option in a `CHOICE`
- A number that represents the position of an element in a `SEQUENCE` or `SET`

CODE EXAMPLE 9-14 shows how to use navigation strings to extract `Morf` instances from a list.

CODE EXAMPLE 9-14 Using Navigation Strings With the `extract` Function

```
...
#include <pmi/hi.hh>
...
// SatelliteSeq ::= SEQUENCE {
//     name      GraphicString,
//     value     Integer32,
//     checkSum  CheckSum
// }
//
// SatelliteData ::= SET OF SatelliteSeq
//
// If Syntax syn is associated with SatelliteData and
// Morf satSetMf is associated with syn...
...
// satSetMf is a SET OF; try to get the third element in SET
Morf seqMf = satSetMf.extract("3");
if (!seqMf) return(NOT_OK);

// Three ways to get the name from the third element in
// the satSetMf set
Morf nameMf1 = seqMf.extract("name");
Morf nameMf2 = satSetMf.extract("3.name");
Morf nameMf3 = satSetMf.extract("3.1");
...
```

9.4.3 Getting the Value Assigned to a `Morf` Instance

To obtain the `Asn1Value` instance that represents the value assigned to a `Morf` instance, call the `get_value` function on the `Morf` instance.

For list type `Morf` instances, you can only get either the string representation or the `Asn1Value` representation of a `Morf` instance's value. For some scalar `Morf` instances, you can retrieve the value directly into an integer or real variable as explained in Section 9.4.4 "Getting Scalar Values Assigned to a `Morf` Instance" on page 9-29.

You may need to get the value of a `Morf` instance for another function or application that expects an `Asn1Value` instance.

The `get_value` function is also useful for obtaining the real value associated with scalar values such as `ENUMERATED` and `OBJECT IDENTIFIER` values. From the resulting `Asn1Value` instance, you can decode the value (not the identifier) and assign it to an instance of a class such as `GenInt` or `Oid`.

9.4.4 Getting Scalar Values Assigned to a `Morf` Instance

The `Morf` and `Asn1Value` classes provide functions for assigning a scalar `Morf` instance's value directly to a variable of another type as follows:

- The `Morf` class provides functions for retrieving values as `DataUnit`, `long`, `GenInt`, or `double`.
- The `Asn1Value` class provides functions for converting the `Asn1Value` instance associated with a `Morf` instance into another type.

To retrieve the string representation of any scalar value, call the `get_str` function of the `Morf` class. Calling the `get_str` function on a list type `Morf` instance is equivalent to calling the `get` function on that instance.

The other functions of the `Morf` class for scalar values return values that can be assigned to variables of numeric types. These functions are very useful for extracting numeric data because you do not have to convert from a string or `Asn1Value` instance to a number.

TABLE 9-7 lists the functions for obtaining numeric scalar values from a `Morf` instance.

TABLE 9-7 Functions for Extracting Numeric Scalars Into Numeric Types

Function	Description
<code>get_dbl</code>	Returns the numeric value assigned to the <code>Morf</code> instance as a double. If the <code>Morf</code> instance is not a scalar nor based on a numeric type, <code>get_dbl</code> returns 0.0.
<code>get_gint</code>	Returns the integer part of the numeric value assigned to a <code>Morf</code> instance as an instance of the <code>GenInt</code> class. The <code>GenInt</code> class eases the handling of arbitrary integers. If the <code>Morf</code> instance is not a scalar nor based on a numeric type, <code>get_gint</code> returns 0.
<code>get_long</code>	Returns the integer part of the numeric value assigned to a <code>Morf</code> instance as a long. If the <code>Morf</code> instance is not a scalar nor based on a numeric type, <code>get_long</code> returns 0.

These functions are valid only for values the base type of which is `BOOLEAN`, `ENUMERATED`, `INTEGER`, `OCTET STRING`, or `REAL`. The following rules govern the conversion of these values to numeric values:

- Nonnumeric values or non-scalar values return 0 (or 0.0).
- `REAL` values retrieved by using the integer functions return only the integer part of the `Morf` instance's value. For example, `get_gint` and `get_long` return 6 for 6.022e23.
- `BOOLEAN` values return 0 (or 0.0) for `FALSE`, and nonzero for `TRUE`.
- `ENUMERATED` values return the integer value associated with the named value.
- `OCTET STRING` values are returned as a single number. For example, `get_long` converts a string of four octets to an integer with four significant bytes, that is, an integer in the range 0 to 2^{31} ($0..2^{31}$).

In addition to numeric values or strings, you may want to extract a `Morf` instance's data directly into other types. To get other types of data, you can use `get_type` to get an `Asn1Value` instance of the value, then call one of the decoding functions on the `Asn1Value` instance. It is often easier to work with data in a closely matching type, and sometimes it may be required. For example, if a function takes an `Oid` instance as a parameter, you may want to extract an `Oid` instance directly from a `Morf` instance of type `OBJECT IDENTIFIER`.

Using the decoding functions of the `Asn1Value` class makes it easy to extract data into the following types:

- Boolean
- Octet
- OID

TABLE 9-8 lists the functions of the `Asn1Value` class for decoding data and the conversions they perform.

TABLE 9-8 Functions of the `Asn1Value` Class For Decoding Data

<code>Asn1Value</code> Class Function	Source ASN.1 Type	Return Values
<code>decode_bits</code>	<code>BIT STRING</code>	<code>decode_bits</code> returns: <ul style="list-style-type: none"> • A string of octets that represents the <code>BIT STRING</code> value • The number of octets in the string If necessary, the beginning of the string is padded with zeroes to make an octet.
<code>decode_boolean</code>	<code>BOOLEAN</code>	<code>decode_boolean</code> returns a boolean value.
<code>decode_octets</code>	<code>OCTET STRING</code>	<code>decode_octets</code> returns: <ul style="list-style-type: none"> • The octets of the <code>OCTET STRING</code> value • The number of octets returned
<code>decode_oid</code>	<code>OBJECT IDENTIFIER</code>	<code>decode_oid</code> returns an instance of the <code>Oid</code> class that represents the <code>OBJECT IDENTIFIER</code> value.

To use these functions on data stored in a Morf instance, you must first call `get_value` to get the `Asn1Value` instance of the Morf instance.

CODE EXAMPLE 9-15 shows how to use `decode_oid` to retrieve an instance of the `OBJECT IDENTIFIER` type from a Morf instance.

CODE EXAMPLE 9-15 Decoding a Morf Instance Directly Into an Oid Instance

```
...
#include <pmi/hi.hh>
...
// TopoNodeAttrList ::= SEQUENCE {
//     view      TopoNodeId,
//     attrs     SET OF OBJECT IDENTIFIER
// }
//
// Assume morf is a Morf associated with the Syntax of
// TopoNodeAttrList... extract each OBJECT IDENTIFIER
// in attrs as an instance of the Oid class.
...
Morf attrMf = morf.extract("attrs"); // Get the attrs SET
if (attrMf.num_elements() > 0) {
    Array(Oid) oids(attrMf.num_elements());
    Array(Morf) oidsMf = attrMf.split_array();
    Result r;
    for (int i=0; i<oidsMf.size;i++) {
        // Need Asn1Value to get Oid from Morf
        r = oidsMf.get_value().decode_oid(oids[i])
        if (r == NOT_OK) oids[i] = Oid(DU("")); // NULL
    }
    ...
    // Do something useful with all those OIDs
    ...
}
...
```

9.5 Using the MorfBuilder Class

The `MorfBuilder` class makes it easier to work with very complex `Morf` instances, especially if the ASN.1 syntax includes combinations of `CHOICE`, `SET`, or `SEQUENCE`.

The `MorfBuilder` class does not ease the decoding of `Morf` data, but it is possible to extract any part of a `MorfBuilder` instance as a new `Morf` instance. You can then use techniques described in the following sections to parse and decode the `Morf` instance:

- Section 9.3 “Parsing Complex ASN.1 Values” on page 9-9
- Section 9.4 “Decoding Complex ASN.1 Values” on page 9-24

To build complex `Morf` instances by using only the `Morf` class, you must either build very complex string representations of the data or carefully assemble subcomponents into larger arrays and build a new `Morf` instance from the array.

Use the `MorfBuilder` class for:

- Constructing a `MorfBuilder` instance
- Adding data to a `MorfBuilder` instance
- Selecting a syntax for `CHOICE` values
- Setting a navigation type for `SEQUENCE` values
- Validating the data in a `MorfBuilder` instance
- Assembling the `MorfBuilder` data into a single `Morf` instance

The `MorfBuilder` class enhances the features of the `Morf` class. Therefore you need to understand how to use the `Morf` class to be able to work with `MorfBuilder` instances. You will need to navigate the structure of the `MorfBuilder` instance’s underlying syntax by using navigation strings. Refer to Section 9.4.1 “Getting a String Representation of a `Morf` Instance” on page 9-25 for more information about how syntactic structure is represented in a string.

9.5.1 Constructing a MorfBuilder Instance

To construct a `MorfBuilder` instance that corresponds to any `Syntax` instance, use the one of the constructors of the `MorfBuilder` class listed in TABLE 9-9.

TABLE 9-9 Constructors of the `MorfBuilder` Class

Constructor	Description
<code>MorfBuilder(Morf& morf)</code>	Constructs a <code>MorfBuilder</code> instance by using the syntax and values stored in an existing <code>Morf</code> instance
<code>MorfBuilder(Syntax& syntax)</code>	Creates a <code>MorfBuilder</code> instance based on the syntax of a <code>Syntax</code> instance with no assigned value
<code>MorfBuilder(CDU attr_name, Platform& plat = Platform::def_platform)</code>	Looks up the syntax of <code>attr_name</code> on the named <code>Platform</code> instance or the default <code>Platform</code> instance and uses it to create a new <code>MorfBuilder</code> instance
<code>MorfBuilder(const MorfBuilder& old_mbd)</code>	Creates a new <code>MorfBuilder</code> instance that has the same syntax and values as an existing <code>MorfBuilder</code> instance

9.5.2 Adding Data to a MorfBuilder Instance

Adding data to a `MorfBuilder` instance assigns values to members contained in the `MorfBuilder` instance. You can assign a value to any member of a `MorfBuilder` instance in any order. To assign a value, call one of the following functions of the `MorfBuilder` class:

- `set`, specifying a string representation of the value you want to assign
- `set_raw`, specifying a `Morf` instance corresponding to the syntax and the values you want to assign to the member

These functions replace the value of the `MorfBuilder` instance (or the selected member) and any contained instances with new values. Any updates you have made to the selected member or the members it contains will be lost unless they have been cached.

To update the currently cached image of a `MorfBuilder` instance or any of its members, call `validate` or `get_raw` before calling `set` or `set_raw`. These functions update the cached copy of the `MorfBuilder` instance. After you call `set` or `set_raw`, call `validate` or `get_raw` again to update the cached copy with the latest changes.

Section 9.5.5 “Validating the Data in a MorfBuilder Instance” on page 9-38 and Section 9.5.6 “Assembling MorfBuilder Data Into a Single Morf Instance” on page 9-39 contain more information about the internally cached `MorfBuilder` instance.

The `set` function of the `MorfBuilder` class is similar to the `set` function of the `Morf` class, but it also takes an optional navigation string. Use the navigation string to specify the attribute or member of the `MorfBuilder` instance to which to assign the value. Refer to Section 9.2.1 “Creating a Morf Instance From String Data” on page 9-2 for details on forming the value string.

The `set_raw` function uses a `Morf` instance instead of a string to assign a value to a `MorfBuilder` instance or one of its members.

CODE EXAMPLE 9-16 shows how to update members of a `MorfBuilder` instance.

CODE EXAMPLE 9-16 Using `set` to Update a `MorfBuilder` Instance

```
...
#include <pmi/hi.hh>           // High-level PMI
#include <extpmi/exthi.hh>     // Extended PMI (MorfBuilder, for
                               // example)

...
// SatelliteSeq ::= SEQUENCE {
//     name      GraphicString,
//     value     Integer32,
//     checkSum  CheckSum
// }
//
// If Syntax syn is associated with SatelliteSeq...
...
Result r;
DU name, value, platform;
MorfBuilder mfb(syn);
...
// ...code to retrieve name, value, and checkSum
// from some data source...
...
mfb.set("name", name);
// validate() updates the cached mfb with the new name so
// subsequent set() calls do not clobber this change
if (mfb.validate() == NOT_OK) {
    cout << mfb.get_error_string() << endl;
}
mfb.set("value", value);
if (mfb.validate() == NOT_OK) {
    cout << mfb.get_error_string() << endl;
}
```



```
mfb.set("checksum", checksum);
if (mfb.validate() == NOT_OK) {
    cout << mfb.get_error_string() << endl;
}
...
```

9.5.3 Selecting a Syntax for CHOICE Values

You can select the ASN.1 syntax to use for any `CHOICE` value contained in the `MorfBuilder` syntax in one of the following ways:

- `select_choice` uses a string to name the value selected.
- `set_syntax` uses a `Syntax` instance that corresponds to the value selected.

The function you use depends on what data is available to your program when the `CHOICE` syntax needs to be selected.

- If you know the attributes listed in the `CHOICE` syntax ahead of time, use `select_choice`.
- If you are working with data derived from the platform, you may not know what choices are available to a given instance at run time. At run time, for example, you may be parsing another `Morf` instance to determine which syntax to select. In that case, use `set_syntax` with an instance derived from the `Morf` instance's `get_syntax` function.

You cannot assign values to a member of type `CHOICE`. First, you must use one of these functions to choose the actual syntax. Both functions for selecting a syntax for a `CHOICE` accept navigation strings, so both can be used to set a choice for any member of a constructed type.

Note – Both `select_choice` and `set_syntax` change the `Syntax` instance associated with a member of a `MorfBuilder` instance. After calling one of these functions, the syntax will no longer represent a `CHOICE` value. Calling one of these functions after a syntax has already been chosen by a previous call will return `NOT_OK`.

CODE EXAMPLE 9-17 shows how to use both functions for selecting a CHOICE value.

CODE EXAMPLE 9-17 Selecting a Syntax For a CHOICE Value

```
...
#include <pmi/hi.hh>
#include <extpmi/exthi.hh>
...
// Syntax hostSyn is associated with this ASN.1:
//
//   myHostEntry ::= SEQUENCE {
//       myHostname GraphicString,
//       myIpAddr   CHOICE {
//           ipString  GraphicString, -- "111.222.111.222"
//           ipInt     INTEGER,       -- 32-bit int
//           ipOctet   OCTET STRING(SIZE(4))
//       }
//   }
//
// Syntax ipOctetSyn is associated with the syntax of ipOctet
...
Result r;
MorfBuilder host1Mfb(hostSyn);
MorfBuilder host2Mfb(hostSyn);
r = host1Mfb.set("myHostname", DU("mailhost"));
// handle if r != OK
r = host2Mfb.set("myHostname", DU("newshost"));
r = host1Mfb.validate();
r = host2Mfb.validate();

// SELECT_CHOICE()
// Set host1Mfb's myIpAddr to use ipString, then assign value
r = host1Mfb.select_choice("myIpAddr", "ipString");
// myIpAddr is now a GraphicString type
r = host1Mfb.set("myIpAddr", DU("111.222.111.12"));
r = host1Mfb.validate();
...

// SET_SYNTAX
// Set host2Mfb's myIpAddr to use ipOctet;
r = host2Mfb.set_syntax("myIpAddr", ipOctetSyn);
Morf ipOct(ipOctetSyn);
if (!ipOct) {
    // handle error
}
```

CODE EXAMPLE 9-17 Selecting a Syntax For a CHOICE Value *(Continued)*

```
ipOct = ipOct.set(DU("6FDE6F0D")); // 111.222.11.13
r= host2Mfb.set_raw("myIpAddr", ipOct);
...
```

9.5.4 Setting a Navigation Type for SEQUENCE Values

You can control whether navigation strings address members of a sequence by names or by position number. By default, you must use the name of a member of a sequence in a navigation string.

Instances of the `MorfBuilder` class have a property called `access_type` that indicates how members of a `SEQUENCE` must be addressed. The `access_type` property can be set for any component of a `MorfBuilder` syntax that represents a `SEQUENCE` type.

The `access_type` property can be set to one of the following:

- `by_name` (default) means that you must use the attribute name to refer to members of a `SEQUENCE`
- `by_index` means that you must use a position number to refer to members of a `SEQUENCE`

To get the value of the `access_type` property for a `SEQUENCE` component of a `MorfBuilder` instance, call the `get_prop` function on the `MorfBuilder` instance.

In the call to `get_prop`, specify:

- **A key.** The only valid key is `access_type`. If the selected component is a `SEQUENCE` value, `get_prop` returns either `by_name` or `by_value`. Otherwise, `get_prop` returns a `NULL DataUnit` instance.
- **Optionally, a navigation string.** Use a navigation string to select a component of a constructed value.

To set the `access_type` for a `SEQUENCE` component, use `set_prop`. Use a navigation string to select a component of a constructed value.

CODE EXAMPLE 9-18 shows how to use `get_prop` and `set_prop`.

CODE EXAMPLE 9-18 Using `get_prop` and `set_prop`

```
...
#include <pmi/hi.hh>
#include <extpmi/exthi.hh>
...
```

CODE EXAMPLE 9-18 Using `get_prop` and `set_prop` (Continued)

```
// MorfBuilder mbd is associated with the simple syntax:
//
//   SEQUENCE {
//       int INTEGER,
//       char OCTET STRING
//   }
//
...
Morf morf;
Result r;
...
// using get_prop()
if (mbd.get_prop("access_type") == DU("by_name")) {
    morf = mbd.get_raw("char");
} else if (mbd.get_prop("access_type") == DU("by_index")) {
    morf = mbd.get_raw("2");
} else {
    cout << "No access_type available!" << endl;
}
...
// using set_prop()
r = mbd.set_prop("access_type", "by_name");
if (r == OK) morf = mbd.get_raw("char");

r = mbd.set_prop("access_type", "by_index");
if (r == OK) morf = mbd.get_raw("2");
...
```

9.5.5 Validating the Data in a MorfBuilder Instance

At any point during the construction of a MorfBuilder instance, you can validate the values assigned to any component of the underlying syntax. To validate the values assigned to a component, call the `validate` function of the MorfBuilder class.

The `validate` function verifies that the internal values of the MorfBuilder instance or component are valid. If all of the assigned values are valid, the `validate` function returns OK.

The `validate` function accepts a boolean parameter. If you set this parameter to `TRUE`, `validate` only ensures that the values are valid for the type defined in the ASN.1 syntax without making sure that they are tagged with the proper attribute type. For example, an integer value might be valid for an `ENUMERATED` type even though the value is not explicitly tagged as `ENUMERATED`.

The `validate` function updates the internally cached image of the `MorfBuilder` instance with any new changes made since the last update. Call `validate` as you assemble a `MorfBuilder` instance to update the cached image of the instance so that later calls to `set` and `set_raw` do not overwrite previous changes.

9.5.6 Assembling MorfBuilder Data Into a Single Morf Instance

After you have assigned all the values to a `MorfBuilder` instance, extract the data as an instance of the `Morf` class so that you can do something useful with it. To extract the entire `MorfBuilder` instance or any component as a new `Morf` instance, call the `get_raw` function.

Use `get_raw` and a navigation string to extract any part of a constructed value.

If `get_raw` fails to construct a `Morf` instance from the underlying `MorfBuilder` instance, it returns a null `Morf` instance.

The `get_raw` function accepts a boolean parameter.

- If this parameter is set to `TRUE`, `get_raw` attempts to update the cached internal `Syntax` and `Asn1Value` relationships and build a new `Morf` instance. If the `Syntax` instance associated with a `MorfBuilder` instance is a constructed type, changes you make to any member values are not updated in the internally cached copy until you call `get_raw(TRUE)` or `validate`.
- If this parameter is set to `FALSE`, `get_raw` returns the internal cached representation of the `Syntax` and `Asn1Value` assignments. This copy represents whatever data was assembled from the previous call to `validate` or `get_raw(TRUE)`. Any changes made since the last update to the internal cache are not reflected in the `Morf` instance that `get_raw(FALSE)` returns.

Developing Object Behaviors

The Object Development Tools (ODT) of Solstice EM provide a simple and automated framework for adding and writing behaviors for managed objects residing in the Solstice EM MIS. You can define objects and their behaviors by using GDMO and ASN.1.

The GDMO definitions formally specify the syntaxes of attributes, actions and notifications. This defines the interface to the object, which is formally specified using ASN.1. The state changes that the object undergoes as a result of the action or operation on the object or as a result of internal changes (perhaps resulting in the creation of notifications) are specified in an informal way in GDMO english text as object behavior. When implementing an object, the developers need to translate this text into a formal state machine in a programming language such as C++.

This chapter explains how to use the Solstice EM object development tools (ODT) to develop object behaviors.

- Section 10.1 “ODT Overview” on page 10-2
- Section 10.2 “Object Interfaces” on page 10-3
- Section 10.3 “Object Development Overview” on page 10-5
- Section 10.4 “Object Code Generator Utility” on page 10-8
- Section 10.5 “Implementing GDMO Specified Object Behavior” on page 10-14
- Section 10.6 “Debugging Objects” on page 10-26
- Section 10.7 “Generated Files” on page 10-29
- Section 10.8 “TRY Exception Macros” on page 10-32
- Section 10.9 “Object Development Examples” on page 10-34
- Section 10.10 “Object Development Scenario Using Chai Object” on page 10-49

10.1 ODT Overview

The ODT allows you to perform the following operations:

- Generate code and interfaces required to support object behavior for a GDMO/ASN.1-defined object.
- Add or remove *attributes* from the GDMO definition and re-generate the code and interfaces.
- Add or remove *actions* from the GDMO definition and re-generate the code and interfaces.
- Add or remove *notifications* from the GDMO definition and re-generate the code and interfaces.
- Switch from using default behaviors to API-user-extended behaviors without re-generating the code and interfaces.
- Add or remove *discriminators* from the GDMO definition and re-generate the code and interfaces.
- Specify persistence or volatility for the attributes of an object class.
- Create and initialize an instance of a new object after the GDMO for it has been loaded.

The ODT lets you define behavior for actions or define behavior for generating any event, not just the standard ones. ODT also lets you define behavior when attributes are accessed or when object instances are created and deleted.

10.1.1 Supporting Functions

To provide a useful object implementation, Solstice EM provides the following supporting functions:

- Ability to invoke operations on an object
- Response/error generation
- Standard event generation
- Transaction management
- Lock management
- Persistence
- Concurrent access to objects
- MIT management
- Scoping and filtering support
- Validation of user requests (For example, DELETE, GET, SET, and CREATE operations for an object follow rules defined in the GDMO.)
- Automatic instance naming

These supporting functions are hidden within the Solstice EM platform and are used transparently by the user-defined implementation code generated by ODT.

10.1.2 Object Development Components

FIGURE 10-1 shows the major components and interfaces the ODT provides or uses:

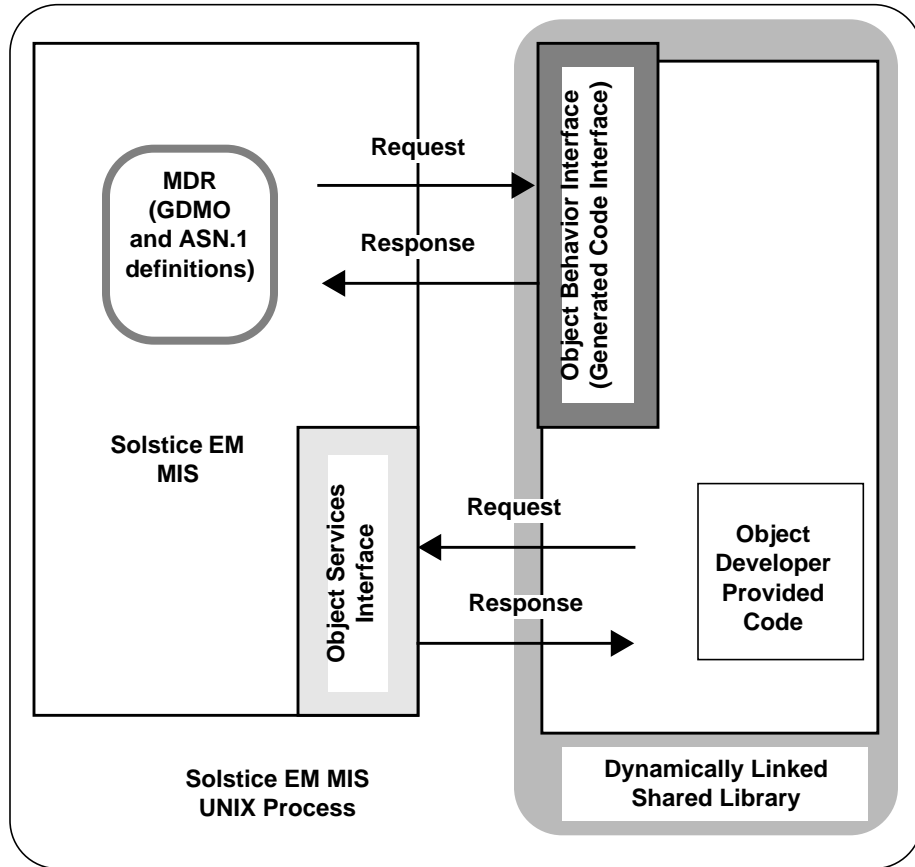


FIGURE 10-1 ODT components

10.2 Object Interfaces

The ODT provides object interfaces for object developers to use when implementing agent or manager-role behaviors using the ODT. The object interfaces are:

- Object Behavior Interface (OBI)
- Object Services Application Programming Interface (OSAPI)

10.2.1 Object Behavior Interface

The Object Behavior Interface (OBI) provides functions that allow the MIS to invoke (or request) object behavior functions developed by an API user and receive responses from the user developed functions.

It provides the following functions:

- Attribute access implementing behavior for GET and SET CMIP operations
- Action access implementing CMIP ACTION operation
- Instantiation access implementing behavior for CREATE, DELETE CMIP operation
- Notification behavior implementing event generation and behavior to be executed on receipt of events

A large part of the software in this interface is *generated* code. This interface also contains generated *stub function interfaces* (also referred to as *stubs*) where you can add your own object behavior code. To write unique object behavior code, you may use the Object Services API or write your own C++ code.

Generated stub function interfaces are provided for the following:

- Each attribute defined for a GDMO object class
- Each action defined for a GDMO object class
- Handling the receipt of notifications by an object
- Handling special instantiation and deletion behavior defined for a GDMO object class

You are required to provide functionality for only the action stub function interface and not for the other stubs functions interfaced. If you do not provide functionality, default behavior functionality is used.

10.2.2 Object Services API

The Object Services API (OSAPI) lets you access services provided by the MIS to implement inter-object behaviors or specialized behaviors. Framework utilities provide capabilities for building, loading, unloading, and instantiating objects.

The decision to use these services depends on the behavior defined for an object. For example, if an action defined for a GDMO object requires the object to check the administrativeState of the log object as part of the action, the user-developed behavior code needs to use the object services interface to issue a get request to obtain the value of the administrativeState attribute of the log object.

The OSAPI provides the following set of services that can be used within an object implementation:

- Issue a get request and (asynchronously) receive any responses

- Issue a set request and (asynchronously) receive any responses
- Issue a create request and (asynchronously) receive any responses
- Issue a delete request and (asynchronously) receive any responses
- Issue an action request and (asynchronously) receive any responses
- Issue an unconfirmed event report request

10.3 Object Development Overview

The process for defining and implementing object behavior is as follows:

1. Define object classes.

Define and develop Managed Object Class (MOC) using GDMO and ASN.1 definitions to include the behaviors for the MOC. If you have existing GDMO and ASN.1 documents that define the appropriate behaviors, you can use those existing files.

2. Compile and load MOC into MDR.

Use the GDMO/ASN.1 compiler to compile and load the GDMO and ASN.1 files into the Meta-Data Repository (MDR).

3. Generate object code and develop behavior.

Use the Object Code Generator (OCG) utility to generate the default object implementation for the MOC. The OCG generates function stub interfaces, a Makefile, object loading and unloading utilities, an object instantiation program, and a README file that contains instructions about the generated files and how to extend the default implementation.

To develop additional behavior, add C++ code at insert areas clearly identified in the generated code. The code you add implements behaviors that are defined in GDMO for the MOC.

Refer to section_____

4. Compile and build object implementation source.

Use the OCG-generated `Makefile.className` to build default or user-extended object implementation. The Makefile builds the object implementation as a dynamic library and a PMI client program for instantiating the object.

5. Load object implementation and restart MIS.

Use the object loading utility, `className.load`, to load the new object implementation into the platform. Then, restart the MIS (using `em_services -start`) to read the new object implementation. When the MIS restarts, the new object implementation is

loaded dynamically into the `em_mis` process. Subsequent CMIP operation on an object instance for this object class results in executing the behavior implemented by the user.

6. Debug object implementation [optional]

User-implemented behaviors might contain errors which result in operation failures and, in some instances, MIS crashes. You can use a debugger to attach to the running MIS or directly debug `em_mis` to debug new object implementations. Using `em_debug`, you can enable or disable developer-provided object-operation traces at runtime.

For a complete scenario that illustrates this process, see Section 10.10 “Object Development Scenario Using Chai Object” on page 10-49.”

10.3.1 Possible Errors

Errors can occur in the following phases of object behavior definition:

■ GDMO object class definition

The GDMO and ASN.1 compiler identify syntax errors in your GDMO and ASN.1 documents. For the Object Code Generator to generate appropriate code, you must provide complete and syntactically correct GDMO and ASN.1 object definition.

■ GDMO object class composition

When the MIS restarts, the object class definition and behavior definition are composed and registered. Any errors in this phase display on your screen.

■ Object class instantiation

When an instance of a class is created, any problems in creating an instance arising out of an improper object definition are returned as an error for the create request.

■ User-developed code

If you add any user-defined code to the generated code, you might introduce errors.

10.3.2 Sanity Check Procedure

It might not be possible to detect all GDMO or ASN.1 errors using the GDMO/ASN.1 compiler or the object implementation process, for example, OID registration clashes, name binding and attribute mismatches for initial values, default values, and so on. Sometimes, late in your object development process, you may find errors or failures that result from errors in the GDMO or ASN.1 definition for the MOC. Use the following sanity-check procedure to minimize potential problems:

1. Comment out ACTION definitions in GDMO.

Comment out the ACTION definitions in the GDMO definition for the object class. You must do this because you cannot compose an object class that contains actions without loading the appropriate action implementation in a dynamic library.

2. Compile and load object class in MDR.

The object class definition in GDMO and ASN.1 must be compiled and loaded in the MDR using the following commands:

```
hostname% em_gdmo -v -f -o /var/opt/SUNWconn/em/usr/data/MDR/  
className.gdmo  
hostname% em_asn1 -v -o /var/opt/SUNWconn/em/usr/data/ASN1/  
className.asn1
```

3. Compose object class.

Use the compose program to compose the new object class. This verifies the OIDs, attributes, and syntax and catches such errors as clashes with existing classes, attribute mismatches, and invalid syntax (referring to a different document/syntax label that is valid but not actually desired by the object implementor). Use the following command:

```
em_compose_oc className
```

Note – If you find errors in this step, you can often get additional error details by using the `oammsg*` and `mdr*` tracing flags with the `em_debug` utility.

4. Load name bindings.

Use the load name binding utilities to load the defined name bindings in the platform. This detects possible errors in the name binding or naming attribute. Use the following command:

```
em_load_name_bindings Namebinding
```

Repeat this for all name bindings specified in the GDMO definition for the object class.

Note – If you find errors in this step, you can often get additional error details by using the `oammsg*` and `mdr*` tracing flags with the `em_debug` utility.

5. Create an instance.

Use OBED or a simple PMI program to create an instance of the MOC. This ensures that the GDMO/ASN.1 definitions are correct and that all CMIP operations can be performed. After you verify this, use OBED or the PMI program to delete the instance.

6. Restore ACTION definitions in GDMO.

Remove the comments to the ACTIONS in the GDMO definition for the MOC. You should now be ready to use ODT.

7. Remove old definitions and prepare to load new object.

Run `em_services -reload` to reinitialize the MDR and MIS. Follow the object development process (see Section 10.3 “Object Development Overview” on page 10-5”) to load your new implementation.

10.4 Object Code Generator Utility

The Object Code Generator utility (OCG) provides a set of C++ classes and methods that you use can to implement the behavior for managed objects defined in GDMO. The OCG is external to the MIS. The code generated and the user-defined implementation reside in a dynamic shared library linked to the MIS.

The OCG generates the C++ stubs for attribute access, instance access, and action access for the class. You fill in the behavior in the stubs. The utility hides the process by which user-defined behavior is connected to the framework. In other words, you only change code stubs for:

- Attribute access (CMIP GET and SET)
- Action access (CMIP ACTION)
- Instance access (CMIP CREATE and DELETE)
- Notification emission (CMIP NOTIFICATION)
- Discriminator-match stub to implement behavior when a discriminator matches, if the user includes the discriminator package in the class definition.

The generated code also contains debugging information to help you trace what happens at run time.

Note – The Solstice EM MIS must be running locally to generate implementation.

10.4.1 Generated Code Interfaces

The generated code interface provides a set of generated code stubs that can be used to invoke user developed object behavior functions. The interfaces and underlying code are produced by the OCG, which operates on information in the meta data repository (MDR) and on information in a configuration file. The GDMO and ASN.1 definitions for GDMO object need to be loaded into the MDR prior to generating the code and interfaces.

The Object Behavior Interface is shown in FIGURE 10-2, with the generated code interface highlighted:

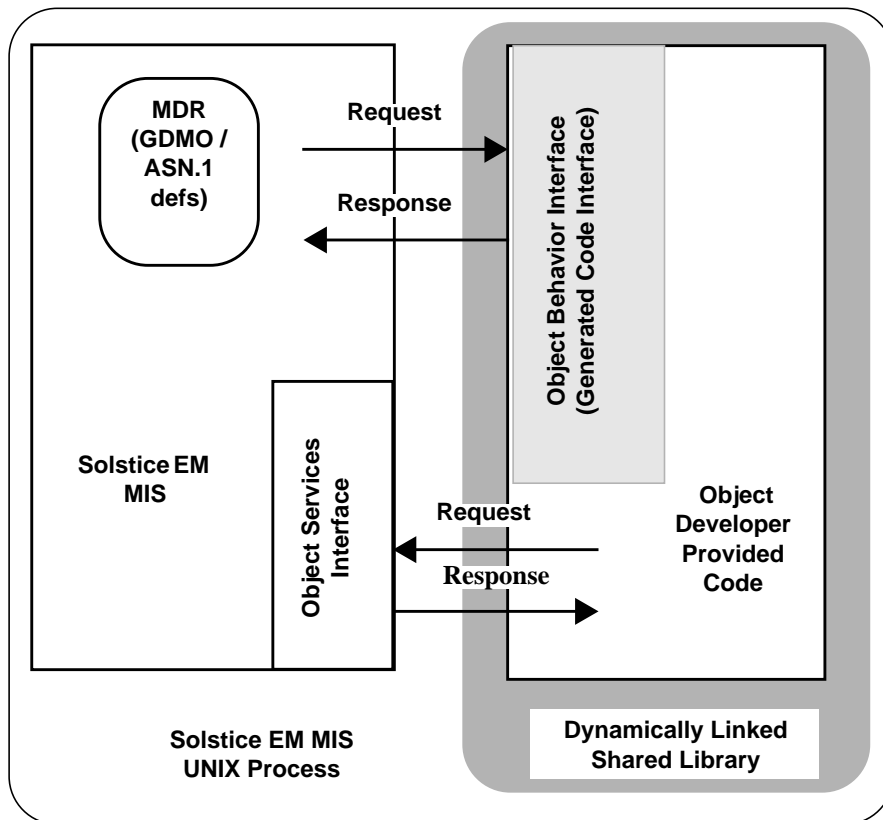


FIGURE 10-2 ODT Framework, with Generated Code Interface Highlighted

10.4.2 Code Generation Components

FIGURE 10-3 shows the components involved in the agent role behavior code generation portion of the Object Behavior Interface. The OCG generates the appropriate agent role behavior code and code stubs for the GDMO-defined managed object class based on the GDMO definition loaded into the MDR and on parameters you specify in a configuration file.

The OCG also generates a PMI client create program that you can use to instantiate an instance of the new managed object class.

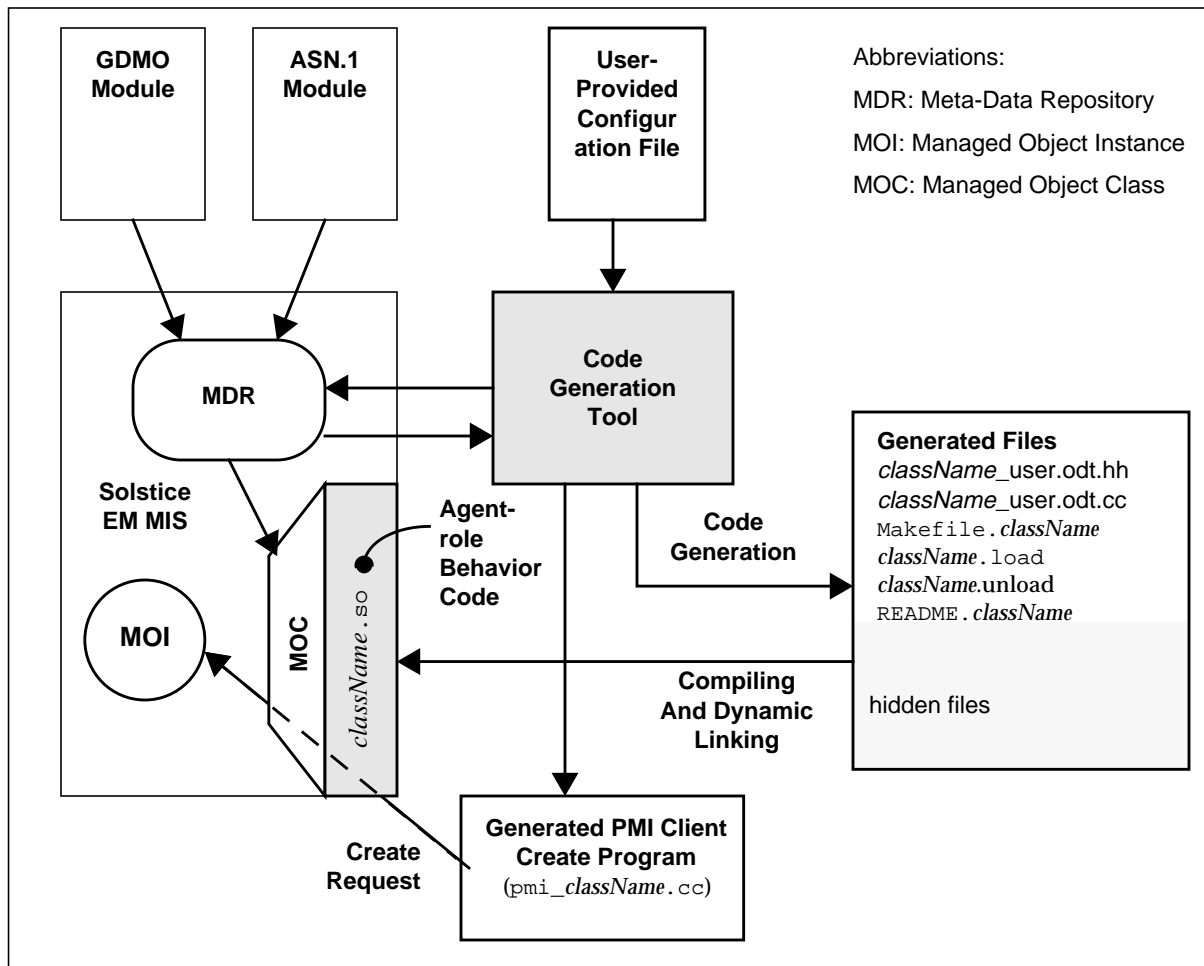


FIGURE 10-3 Code Generation Components

10.4.2.1 Inputs

The Object Code Generator utility takes input from the following sources:

- GDMO and ASN.1 files containing the class descriptions
- Configuration file

10.4.2.2 Outputs

The Object Code Generator utility provides the following output:

- C++ code stubs for attribute access, actions access, and instance access
 - The stub for attribute access allows you to add behavior for each attribute when a CMIP GET/SET is done.
 - The stubs for action access allow you to add behavior for each action supported by the GDMO class definition.
 - The stubs for instance access allow you to add behavior to be executed when CMIP CREATE/DELETE operations are done.
- Code that links the object implementation to the framework

This is called the annotation code. Object implementors should not change any of this code. The annotation OID is unique and is generated automatically.

Note – All the attributes and the object instance are either persistent or volatile. You control volatility on a per-object class basis.

- Makefile that generates an object implementation (shared library)
- Utilities to load and unload object implementation dynamically
- If a GDMO object class has notification definitions, you need to add specialized behavior code to specify when the event should be generated. To do this, you use the `SendEventReportRequest` function provided by the Object Services API (OSAPI).
- GDMO Inheritance is supported. This means behavior defined for a superior class is re-used transparently when generating code for a derived class. You cannot override any behavior inherited from the superior class.

10.4.3 Using the Object Code Generator Utility

Before you use the OCG, you must compile the GDMO and ASN.1 definitions using the GDMO and ASN.1 compiler and restart MIS to load the GDMO and ASN.1 definitions. You then have the following options for the object behavior:

- Default behavior with persistence
- Default behavior with volatile attributes
- Non-default (user-specified) behavior with persistence
- Non-default (user-specified) behavior with volatile attributes

The OCG is a command line function. To run it, use the following command:

```
% $EM_HOME/bin/em_obcodegen -help filename
```

Note – \$EM_HOME is an environment variable used to designate the directory in which Solstice EM is installed, typically /opt/SUNWconn/em.

TABLE 10-1 identifies the options available for em_obcodegen.

TABLE 10-1 OCG Command Line Options

Option	Description
-help	Displays a list of command options.
<i>filename</i>	Identifies the class name to generate code for. The GDMO and ASN.1 files must match this file name.

Example

To generate code for the chai example, you would use the following format:

```
% $EM_HOME/bin/em_obcodegen chai
```

10.4.4 Configuring the Object Code Generator Utility

The specific code the OCG generates depends on a number of configuration parameters. You can define these parameters in any of the following locations:

- In your login shell as user-specific environment variables
- In a local configuration file called `EM_obcodegen.cfg`
- In the global configuration file
`/etc/opt/SUNWconn/em/conf/odt/EM_obcodegen.cfg`

If several developers need to use a standard configuration, use the global configuration file. TABLE 10-2 lists the parameters you can define for ODT configuration.

TABLE 10-2 Object Development Tool Configuration File Parameters

Parameter	Default Value	Description
CODEGENDIR	. (Current directory)	Directory for writing the generated code files.
DATASTORAGE	PERSISTENT	Data storage for the object class. Valid values are VOLATILE or PERSISTENT.
OBAPITRACE	YES	Enables runtime functional tracing.
OBAPIDEBUG	YES	Enables runtime debugging output.
HIDDENDIR	.hidden	Indicates where all the hidden annotation and implementation code is generated. Users should not modify files located in this directory.
FILTER_ATTR	DiscriminatorConstruct	If the object class needs to support event discrimination, set this flag to DiscriminatorConstruct. This causes the discrimination secretary to be generated.

10.4.5 How Filter Attributes Affect Code Generation

The GDMO definition for your object class can include following three attributes that affect how code is generated for receiving events:

- DiscriminatorConstruct
- OperationalState
- AdministrativeState

If these attributes exist in your GDMO definition and FILTER_ATTR is set to DiscriminatorConstruct, then OCG generates the following code:

```
receive_event(EventType, EventInfo);
```

If FILTER_ATTR is set to DiscriminatorConstruct and any of these attributes are not defined in your GDMO, then you see a warning message and this line of code is not generated.

10.5 Implementing GDMO Specified Object Behavior

There are 5 important operational aspects to the Generated Interfaces the ODT user must understand:

- MIS object modeling concepts
- Asynchronous interface behavior
- Use of the subfetch, subread, subwrite and substore interfaces
- Propagation of errors
- Serialization of object requests

10.5.1 MIS Object Modeling Concepts

Solstice EM defines object behaviors according to the abstractions described in TABLE 10-3.

TABLE 10-3 Behavior Abstractions

Behavior	Function
Instantiation	object creation and deletion
Containment	management of children
Attribute Management	storage/access to attributes
Actions	action behavior

These four areas have C++ base classes that define interfaces for the functionality listed above. The base C++ classes that implement these behaviors are known collectively as Secretaries. A Secretary is a named component of code that implements a specific functionality. New object behaviors are added to the MIS framework by describing to the MIS the set of secretaries (components of behavior) that will implement the behaviors for a particular GDMO Object Class. This description is called the Object Behavior Definition (OBD). The OBD is a list of the secretaries and is automatically generated by the ODT and is hidden from the ODT user.

The framework by default provides for instantiation, containment, attribute and action support of agent role behavior. The ODT user need only insert behavior code at well defined interfaces to supplement or augment these default behaviors.

New behaviors are created by deriving new C++ object classes from the base C++ secretary classes. The ODT generates these derived classes on behalf of the user for a specific GDMO Managed Object Class. To simplify the problem of creating agent behaviors the ODT has exposed the interfaces listed below.

The following list summarizes the functions or points in the generated code that can be modified by a user of the Object Behavior Interface. For most objects, you will **not** need to supply additional code for every interface point listed here:

- Attribute secretary read function: *className_AttrSecty::read*
- Attribute secretary write function: *className_AttrSecty::write*
- Attribute secretary read function: *className_AttrSecty::fetch*
- Attribute secretary write function: *className_AttrSecty::store*
- Action secretary check function: *classNameActionSecty::action*
- Action secretary perform function: *className_InstanceSecty::create_vote*
- Action secretary do not perform function:
className_AttributeSecty::destroy_vote
- Dynamic Loader interface: *className_loader*

10.5.2 Asynchronous Interface Behavior

Asynchronous behavior is introduced at an interface by providing a callback parameter as part of the interface invocation. The expectation is that the implementor of the interface will only invoke the callback when the interface functionality is complete. Solstice EM provides the following asynchronous interfaces:

- *fetch*
- *store*
- *action*

These provide interfaces which allow the user an asynchronous interface to fetch data, store data and perform actions in an asynchronous manner. A callback is invoked by using the `exec` method:

```
cb.exec(parmarater);
```

The parameter to the `exec` invocation is one of the ways used to indicate status to the invoker. For more information on this parameter, see Section 10.5.4 “Propagation of Errors” on page 10-22”.

The Object Framework uses a state machine model to manage the different phases to complete the different CMIS requests. These requests and the interfaces they invoke are described in TABLE 10-4 and TABLE 10-5.

TABLE 10-4 Interfaces for CMIS Requests

Interface	Action
Fetch	Issue fetch for all operations in request and wait for all Callbacks
Read	Issue read for all attributes in request
Write	Issue write for all attributes in request
Store	Issue store for all attributes in request and wait for all Callbacks

TABLE 10-4 lists the order of interfaces invoked for each CMIS request.

TABLE 10-5 Order of CMIS Request Interfaces

CMIS Request	Order of interfaces invoked
M_GET	fetch, read
M_SET	fetch, read, write, store
M_CREATE	write, store

At each phase the framework invokes the appropriate interface for all attributes in the CMIS request. An `M_GET` with an attribute list with two attributes will invoke `fetch` twice followed by 2 `read` requests. An `M_SET` for three attributes will invoke `fetch` for each of the three attributes, followed by a `read` for each of the three attributes followed by a `write` and a `store` for each of the listed attributes.

Consider the `fetch` interface as a simple example of how asynchronous behavior is achieved:

```
ClassName_AttrSecty::fetch(ai, callback)
```

The invoker of this interface does not expect that when the `fetch` function returns that the attribute specified by `ai` has been fetched. The implementation may need to send a request to another entity or agent to fetch the attribute data. However, instead of waiting for the remote entity to respond, the `fetch` implementation can store the callback parameter and schedule its own callback for when the remote entity provides the attribute data.

On receipt of the attribute data, the `fetch` interface then invokes the stored callback, indicating to the originator of the `fetch` invocation that data is now available synchronously. In other words, you can pass a callback to `subfetch` which, when called, calls the original callback passed in `fetch`. The invoker can then invoke the `ClassName_AttrSecty::read` synchronously.

Similarly for storage operation, the `ClassName_AttrSecty::store` function is passed a callback. Only when the data has been stored should the implementor invoke the passed callback.

The Solstice EM Object Framework invokes the generated Object Behavior Interfaces `fetch`, `read`, `write` and `store` in a particular order depending on the CMIS request being performed. These interfaces are designed to allow asynchronous requests to be satisfied. The `fetch`, `store` and `action` interfaces are all asynchronous interfaces. FIGURE 10-4 outlines the normal order of operations for a GET request. The framework calls the `UserSecty` code `fetch` function for all attributes providing as parameters the attribute identifier and the callback to be invoked when the `fetch` has been completed. The `fetch` code is responsible for fetching the attribute data from wherever it is stored.

Once the data has been fetched the `UserSecty` code must then call the callback provided. The framework maintains a count of all the outstanding callbacks. When all callbacks have been received, the framework enters the `read` phase. The assumption is that since the (asynchronous) `fetch` has completed, the (synchronous) `read` can complete without blocking. Remember that `read` and `write` are synchronous operations for accessing memory, while `fetch` and `store` are asynchronous and act on disk storage.

In the sequence diagrams below, half arrows indicate asynchronous operations and full arrows denote synchronous operations.

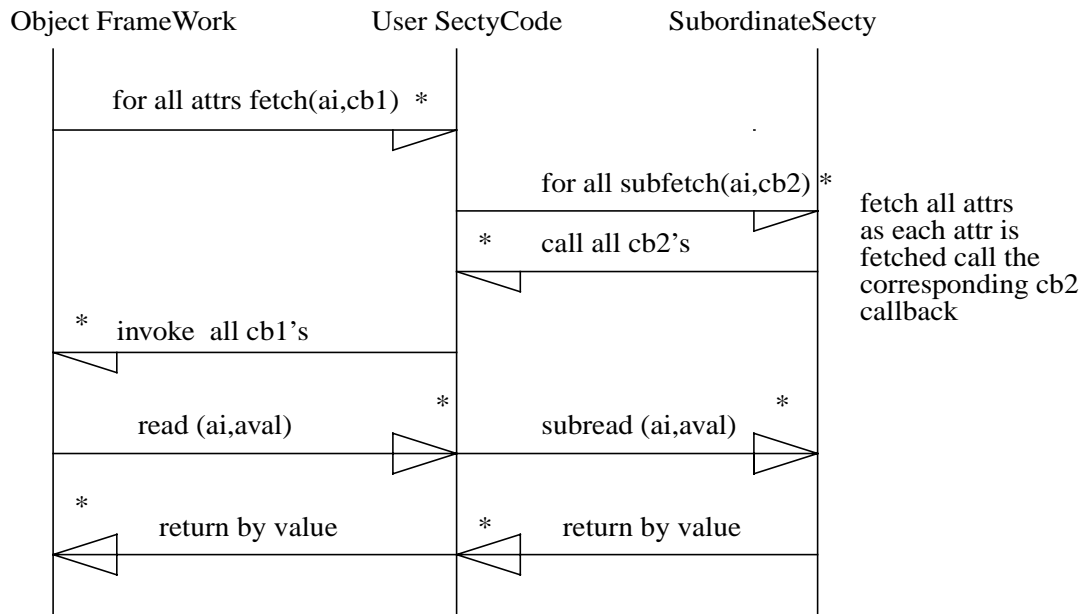


FIGURE 10-4 Sequence Diagram for `M_GET` operation

FIGURE 10-4 indicates the normal flow for the code that is generated by the ODT. In particular it is important to understand that the subordinate operations do not need to be called unless default behavior is desired. However, once a subordinate `fetch` function is called, the subsequent `read` on the `fetch` attribute should also call the `subread` function. The use of the sub functions is detailed in the `sub` operation section following.

The diagrams below outline the sequence flow for `M_ACTION` and `M_SET` operations.

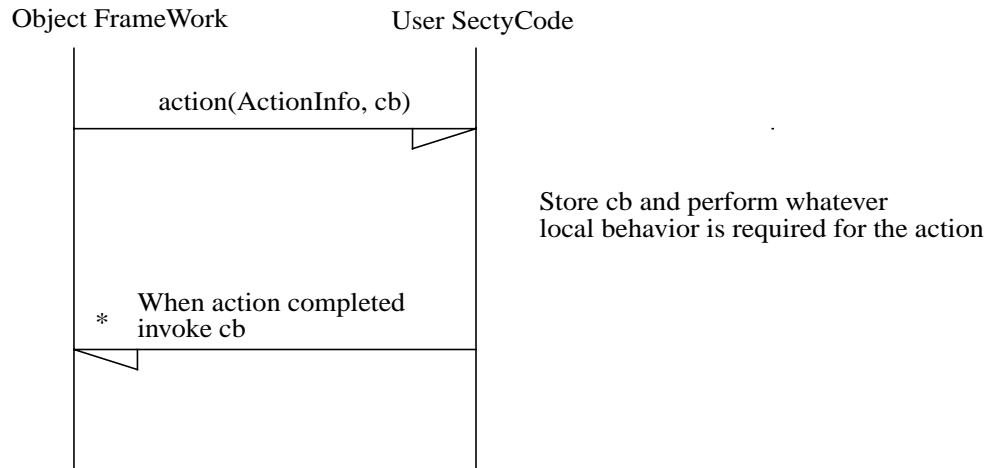


FIGURE 10-5 Sequence Diagram for `M_ACTION`

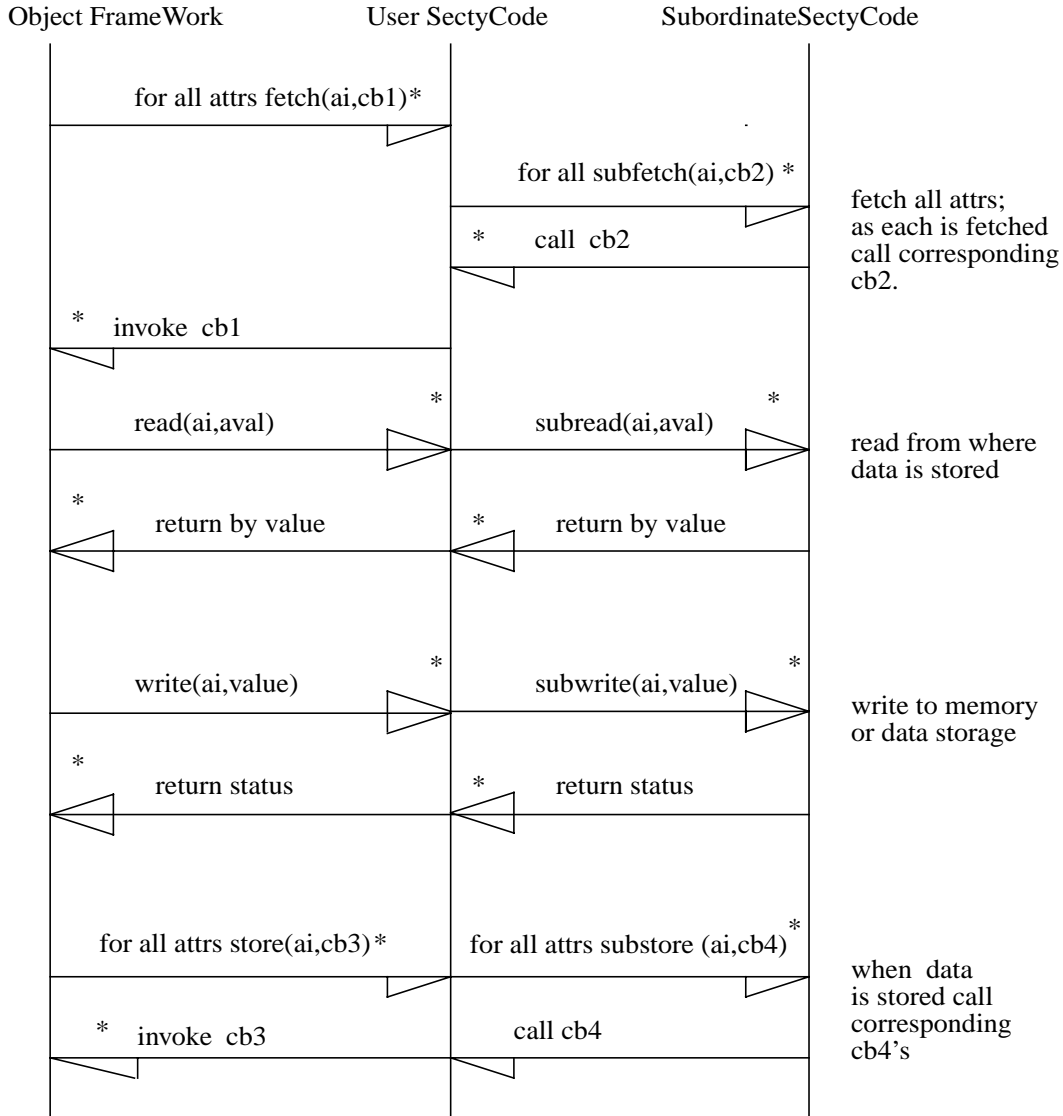


FIGURE 10-6 Sequence Diagram for M_SET

It is important for the ODT interface implementor to obey the rules of the interface. Essentially the implementor must issue a callback for every asynchronous received. Failure to do so will cause the original request to be suspended. The MIS will continue to operate and service other requests

The default behavior for `fetch` and `store` is that the callback is executed immediately from the `fetch` and `substore` operations. If the ODT developer does not wish to use this default behavior and decides to delay the invocation of the `fetch` or `store` callback the ODT developer must register the object class implementation with the `OamAsyncable` Interface. To register an object class in the `OamAsyncTableIf` the `oamasyntblif::AppendAsyncTbl` method is used:

```
oamasyntblif::AppendAsyncTbl(Oid("1.2.3.4.5.6.7", TRUE);
```

This registers Object Class 1.3.4.5.6.7 as an asynchronous class.

Note – Failure to register the Object Class as asynchronous may result in random MIS code dumps.

10.5.3 Sub Operations: `subfetch`, `subread`, `subwrite`, `substore`

The interfaces `subfetch`, `subread`, `subwrite`, and `substore` are used to interface to a service layer coupled to the `UserSecty` code via a late binding mechanism in the MIS framework. The services currently offered in this manner are `Volatile` and `Persistent`. `Volatile` refers to an in-core storage mechanism for managed objects. `Persistent` refers to the persistent service component of Solstice EM. `Volatile` or `persistent` behavior is defined by configuration. See Section 12.5.6 “Configuring the Object Code Generator” for information on how to configure for `volatile` or `persistent` behavior. By obeying the conventions of the sub operations an object class implementation can be switched easily from `volatile` to `persistent`.

To use the `volatile` or `persistent` services the ODT developer must invoke the sub operations. The rules that apply to the `fetch`, `read`, `write` and `store` interface apply again. The semantics are the same.

The invocation of sub operations are not mandatory. However the rules of the interfaces must be obeyed. `Subfetch` must always be called before `subread`. `Subread` cannot be called until the callback passed to `subfetch` has been invoked. `Subwrite` must always be called before `substore`. Data is not deemed to be stored until the `substore` callback has been called.

The default code generated by ODT for `fetch` is to call `subfetch`. The `subfetch` code then invokes the original invoker's callback. If you pass a different callback to `subfetch`, it must call the original callback to `fetch`. If the ODT developer chooses not to call the `subfetch` function then the ODT developer assumes responsibility for invoking the callback supplied as an argument to the `fetch` call.

```
ClassName::fetch(ai,framework_cb)
{
    if ( using_sub_service ) {
        subfetch(ai,framework_cb
    return OK;
    }
    // Set up for my own callback, allocates a Context structure
    // which saves the callback, This context is passed to ClassName::my_cb
    schedule_my_cb(ai,framework_cb)
    return OK;
}
// Local Callback mechanism invoked when Data has been fetched
ClassName::my_cb(Ptr P1, Ptr P2)
{
    MyLocalContext *p_context = (MyLocalContext*) P1;
    // Invoke original callback with 0 as an indication of Success
    p_context->framework_cb.exec((Ptr) 0 );
}
```

A user may choose to only use persistence for a write through capability. In this case the ODT developer would only need to use `subwrite` and `substore`. Once data has been stored to persistence using the `subwrite` and `substore` interfaces, the user may subsequently use `subfetch` and `subread` to retrieve the data.

10.5.4 Propagation of Errors

The ODT framework has three different means to reflect errors back to the request invoker:

- Return Value
- Solstice EM MIS Exceptions
- Operr Data Structure

The interface error mechanisms have been designed to allow for maximum flexibility while at the same time offering support to hide the complexity of building a CMIS error PDU. CMIP errors come in two flavors: List Errors and Fatal Errors.

List errors occur on `M_SET` and `M_GET` operations and indicate a partial error on a specific attribute or list of attributes. For example, an `M_SET` operation on an object containing `AdministrativeState` and `OperationalState` would result in a `SET_LIST_ERROR` response since `OperationalState` is read-only. An attempt to create an instance of an unknown object class would result in a Fatal Error of `NO_SUCH_OBJECT_CLASS`.

The Object Framework determines the appropriate error for errors reflected by return value or exceptions. It builds the error PDU and issues the error response. The `Operr` mechanism allows for complete flexibility to indicate any type of error to the request invoker. The `Operr` mechanism is used at the `fetch` and `store` interfaces only.

Note – The `Operr` method is the recommended means to generate errors for all operations.

10.5.4.1 Return Value

The Return Value is `OK` or `NOT_OK`. The specifics of which error an `OK` or `NOT_OK` produces is detailed in the sections specific to each interface.

10.5.4.2 MIS Exceptions

Exceptions are passed back across an interface by using the `THROW` or `VTHROW` macros. The specifics of which error is produced for an interface is detailed in the sections specific to each interface.

10.5.4.3 Operr Returns Values

The `Operr` class has four constructors which will construct different error responses. The types of error responses that can be constructed are:

- ANY CMIS Error Message
- Processing Failure with Specific Error
- Processing Failure with Probable Cause Oid Format
- Processing Failure with Probable Cause Integer Format

```
Operr(Message * ErrorMessage);
```

This will cause the MIS to send an error message, pointed by `ErrorMsgp` to the original requester. The error message is allocated using the `Message::new_message` method. All fields must be completed in the `Error Message`.

```
Operr(Oid & errorId, Asn1Value &errorInfo);
```

This will cause the MIS to send a `ProcessFailure` message to the original requester. `errorId` and `errorInfo` constitute the specific error part of that `ProcessFailure` message. The syntax for `SpecificErrorInfo` is:

```
SpecificErrorInfo ::= SEQUENCE {  
    errorId          OBJECT IDENTIFIER,  
    errorInfo        ANY DEFINED BY errorId  
}
```

The `errorId` parameter must be a valid OID. The `errorInfo` must be a properly encoded `Asn1Value` as defined by `errorId`.

```
Operr(int OperrInt);
```

This will cause the MIS to send a `ProcessFailure` message to the original requester. The specific error part of the message is set to `probableCause` (2.9.3.2.7.18) and the error number indicated by `OperrInt`.

```
Operr(Oid & OperrOid);
```

This will cause the MIS to send a `ProcessFailure` message. The specific error part of the message is set to `probableCause` (2.9.3.2.7.18) and the error information indicated by `OperrOid`. In this case, the `OperrOid` must contain one of the valid OID defined in `/SUNWconn/etc/gdmo/dmi.gdmo` for `probableCause`.

To use the `Operr` format of error reporting the user must allocate an appropriate `Operr` instance **on the heap** and then pass it to a `fetch` or `store` callback:

```
Operr *p_err = new Operr(2); // Processing Failure ProbableCause 2

fetch_cb.exec(p_err); // Called function will delete the Operr
return OK;

or

Operr *p_err = new Operr("2.9.3.2.0.0.1");
// Processing Failure ProbableCause
// adaptError, see DMI.ASN1 for
// complete ProbableCause Error list

store_cb.exec((Ptr)p_err); // Called function will delete the Operr
return OK;
```

Note – Fetch and store are called for every attribute. The ODT developer must respond to every fetch and store by issuing a callback for every fetch and store received. The object framework examines these callbacks for errors but only uses the first error reported to generate the requested error response. It will discard any subsequent `Operr` responses.

Note – The ODT developer must allocate the `Operr` Data Structure from the heap. The framework deletes the `Operr` structure when it no longer needs it. `Operr` data structures can only be used in `fetch` and `store` Callbacks. They must not be used for the `::Action` result callback.

10.5.5 Serialization of Object Requests

To guarantee consistency within a managed object instance, the object framework employs a simple lock management scheme to lock the object for destructive operations. The locking mechanism locks an object exclusively for `M_SET`, `M_CREATE`, `M_DELETE` and `M_ACTION` requests. Exclusive locks enforce serialized access to the object and force an operation to complete before the next operation is allowed to start. `M_GET` requests are honored using a shared lock level, allowing multiple `M_GETs` to operate on the same managed instance concurrently.

The default lock mode for `M_ACTIONS` is `LOCK_EXCL` (exclusive). This mode requires that an action completes before any other operation can be started. Since `M_ACTION` requests are not necessarily destructive and may execute for a long time, the framework allows the ODT developer to set the lock mode that a specific `M_ACTION` may be executed at.

To override the default lock level for an action the ODT developer must set the desired action lock level using the `oamlockif` interface defined in `oamlockif.hh`. The function `set_action_lock_level` is used to set the action lock level. The following code example shows how to use the `set_action_lock_level` function:

```
Oid ActionOid("2.9.2.3.8.1");  
oamlockif::set_action_lock_level(ActionOid, OAM_LOCK_SHARE);
```

The preceding code example sets the action defined by `ActionOid` to be shared. This will allow `M_GETs` and other shareable `M_ACTIONS` to be performed on the instance supporting these actions while an action is being executed. It will not allow `M_SETs` to be performed while any `M_GETs` to `M_ACTIONS` are in progress.

Note – If `M_GET` operations seem to block indefinitely and the object implementation supports long running actions set the action level appropriately.

10.6 Debugging Objects

10.6.1 Process

1. Find out the process identifier of the running MIS.

```
hostname% ps -eaf | grep mis
```

You should see output similar to the following:

```
root  9324      1 80 08:26:00 pts/12    0:59 em_mis -k
```


2. Run the debugger against the process identifier of the MIS.

```
hostname% debugger - MIS_pid_from_previous_step
```

Make sure you put a blank space between the hyphen and the process identifier. For the output shown in the previous step, you would use the following:

```
hostname% debugger - 9324
```

3. When the debugger comes up, go to the debugger line and open the file `className_user.cc`. This should look similar to the following:

```
(debugger) file chai_user.cc
```

4. Set a breakpoint in the `className_user.cc` file.

```
(debugger) stop in wherever
```

5. Continue.

```
(debugger) cont
```

10.6.2 Dynamic Loading in Solstice EM

OCG generates a default object implementation for a MOC defined using GDMO and ASN.1 and loaded in the MDR. Application developers can modify the default object implementation. The object implementation is built as a shared library that is loaded dynamically into the MIS at startup (`em_services`). Similarly, object implementations can be unloaded dynamically at MIS startup.

ODT provides two utilities that are generated as part of OCG:

- `className.load` installs the shared library and adds it to the system configuration file that MIS reads to load the object implementation.
- `className.unload` removes the shared library and removes it from the system configuration file that MIS reads to unload the object implementation.

Because the object implementations are loaded at different address spaces in the MIS when the MIS is started, an application developer cannot set a breakpoint at a well-known location in the dynamically loaded shared library. To enable users to debug object implementation, `em_mis` provides a well-known breakpoint that you can use before providing other breakpoints in the dynamically-loaded object implementation.

10.6.3 ASN.1 and GDMO Debugging

Solstice EM does not provide specific tools for debugging ASN.1 and GDMO files. For complete information on these syntax definitions, see the following:

- ITU X.208 ISO/IEC 8824, Specification of Abstract Syntax Notation One (ASN.1)
- ITU X.209 ISO/IEC 8825 Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1)
- ITU X.722 ISO/IEC 10165-4, Information Technology—Open systems Interconnection - Structure of Management Information—Part 4: Guidelines for the Definition of Managed Objects (GDMO)

10.6.4 Printing ASN.1 Values in Human-Readable Form

To print ASN.1 values (information in `Asn1Value` form) in human-readable form, first define the following in your `.cc` file:

```
Debug_on (className_info);  
Asn1Value av;
```

Then, use the `print` method of `Asn1Value` defined in the PMI (`asn1_val.hh`) as follows:

```
av.print(className_info);
```

Where:

- `Debug_on(className_info)` is already generated by OCG in `className_user.odt.cc`.
- `Debug_on(className_info)` defines a Debug agent (`className_info`) that can be enabled or disabled at runtime or at compile time by using `Debug_off`, which turns off the agent.

10.6.5 Debugging Flags

OCG generates debug agents (*className_error* and *className_info*) for every object class. If you specify OBAPIDEBUG as YES in the configuration file, OCG enables these agents at compile time. To enable or disable these agents at runtime, use *em_debug* and the following commands:

- *em_debug -c "on className_info"* to enable agent at runtime
- *em_debug -c "off className_info"* to disable agent at runtime
- *em_debug -c "on className_**" to enable all debug agents for *className* at runtime

When debugging behaviors that require you to use Object Services API (for example, if implementing inter-object behaviors), you can enable or disable debug agents specifically for Object Services API calls. To enable these agents, use the "*objsvc_**" options for the *em_debug* utility as follows:

```
$EM_HOME/bin/em_debug -c "on objsvc_test"
$EM_HOME/bin/em_debug -c "on objsvc_error"
```

Or use the following command:

```
$EM_HOME/bin/em_debug -c "on objsvc_*
```

To disable these agents, use *em_debug -c "off objsvc_*. "*

10.7 Generated Files

If you have created valid GDMO and ASN.1 definition files and loaded them into the MDR, when you run OCG it creates the files in the target directory specified in your configuration file:

- *Makefile.className*
- *README.className*
- *className_user.odt.cc*
- *className_user.odt.hh*
- *pmi.className.cc*
- *className.load*
- *className.unload*

For complete examples of each of these files, see Section 10.10 “Object Development Scenario Using Chai Object” on page 10-49.”

10.7.1 Makefile (`Makefile.className`)

You use the Makefile to create (“make”) a dynamic linked library for every object class for either default or user-extended implementation.

10.7.2 Readme File (`README.className`)

The README file explains how to use the files generated by OCG.

10.7.3 User Header File (`className_user.odt.hh`)

This header file contains the class definitions for the object class *className*. OCG generates definitions for the Attribute class, Action class, and Instance class. The Attribute and Action classes contain several helper methods that can be used to access other attributes or actions defined in this file or to perform read/write actions on other attributes or actions defined in this file, while implementing specific behavior for a given attribute or implementing a specific action.

In addition to the class definitions, the user header file defines Action indices, Attribute indices, and OIDs for Attributes, Actions, and Name Bindings.

Note – You are not allowed to modify this file directly. The default object implementation build uses this file, so changes made to it will cause unpredictable results.

To add function prototypes or members in the header file, copy this file to *className_user.hh* and add your code there.

TABLE 10-6 identifies the Attribute classes OCG defines in this file:

TABLE 10-6 Attribute Class Helper Methods

Attribute Class Name	Description
<code>index2AttributeSectyInfo</code>	Converts from <code>AttributeIndex</code> to <code>AttributeInfo</code>
<code>AttributeSectyInfo2index</code>	Converts from <code>AttributeInfo</code> to <code>AttributeIndex</code>
<code>index2ActionSectyInfo</code>	Converts from <code>ActionIndex</code> to <code>ActionSectyInfo</code>
<code>action</code>	Performs action
<code>read/write/fetch/store</code>	Read/Write/Fetch/Store Attribute

TABLE 10-7 identifies the Action classes OCG defines in this file:

TABLE 10-7 Action Class Helper Methods

Action Class Name	Description
<code>index2AttrSectyInfo</code>	Converts from <code>AttributeIndex</code> to <code>AttributeInfo</code>
<code>index2ActionSectyInfo</code>	Converts from <code>ActionIndex</code> to <code>ActionInfo</code>
<code>read</code>	Reads attribute
<code>write</code>	Writes attribute
<code>fetch</code>	Fetches attribute
<code>store</code>	Stores attribute

10.7.4 PMI Client Create Program for Object Instantiation (`pmi_className.cc`)

The PMI client create program file contains PMI client application code that is used to instantiate an instance of the new object class after the dynamic library for the new object class has been linked into the MIS.

10.7.5 User Code File (`className_user.odt.cc`)

The C++ source file contains the user-level methods defined for `Attribute`, `Action`, and `Instance` classes. The user function stubs that OCG generates include: `read`, `write`, `fetch`, `store`, `action`, `create_vote`, and `destroy_vote`. In addition, OCG generates a stub for `receive_event` if discrimination service is used.

Note – You are not allowed to modify this file directly. The default object implementation build uses this file, so changes made to it will cause unpredictable results.

To add user-defined behaviors, copy this file to `className_user.cc` and add your code in the insertion areas clearly identified.

When implementing intra-object behaviors, you should use only the helper methods defined in `Attribute`, `Action`, and `Instances` classes. When implementing inter-object behavior, you should use the Object Services API calls as needed for behavior implementation.

10.7.6 Dynamic Loading File (*className.load*)

When you run this utility, the object implementation build is loaded into the platform as a shared library. The new object implementation is read at MIS startup.

10.7.7 Dynamic Unloading File (*className.unload*)

When you run this utility, the object implementation is removed from the platform. The object implementation is not read at MIS startup.

10.8 TRY Exception Macros

The development environment of Solstice EM includes some exception-handling macros that are used in cases where the C++ compiler does not handle the exception. These exception-handling macros are known as *TRY macros*. The basic elements of the TRY macros are the *TRY block* and the *Handler block*.

10.8.1 Overview

The TRY block brackets the code from which you want to receive exceptions. It must be followed immediately by a Handler block in which you specify how to handle the exception.

Exceptions are scoped dynamically. What this means is that a TRY block establishes a new exception context. When you exit the TRY block, you return to the previous exception context.

10.8.2 Code Structure

The basic structure of a TRY exception is as follows:

```
TRY {
    some block of code that may generate exceptions
}

BEGHANDLERS
    CATCH macros that handle various exceptions
ENDHANDLERS
```

10.8.3 Code Examples

The following example from the chai scenario shows how the TRY macros are used in the generated code:

```
TRY
{
    // Fetch attribute specified by (ai)
    return subfetch(ai,cb);

}
BEGHANDLERS
CATCHALL {

#ifdef ODT_DEFAULT
    return (NOT_OK);
#endif

}
ENDHANDLERS
}
```

10.9 Object Development Examples

Solstice EM comes with several examples that illustrate how to develop object behaviors. All of these examples are located in the `$EM_HOME/src/odt` directory. This directory includes a README file that explains how to build the examples.

TABLE 10-8 identifies the examples and provides a brief description of what each one includes. A complete scenario that illustrates how to develop an object is included in Section 10.10 “Object Development Scenario Using Chai Object” on page 10-49.”

TABLE 10-8 Object Development Examples

Class Name	Description
<code>cellSample</code>	Defines a set of intra-object complex behaviors. Basically, it looks at an object and, if its behavior changes then the behavior of its neighboring objects changes.
<code>chai</code>	Looks at an attribute called “chaiReady” to decide whether there is any chai (tea) ready to drink. If not, it sends an action “brewChai” to make more.
<code>demoPing</code>	Defines behavior of a “native agent.”
<code>demoregistry</code>	Provides an MIS client function to operate as a “remote agent.” This demonstrates how to register an application, similar to a licensing facility.
<code>demoServer</code>	Provides an MIS server function to operate as a “remote agent.” This provides required support for demoregistry and <code>diskInfo</code> examples.
<code>diskInfo</code>	Demonstrates behavior to get information from an external (outside the MIS) process.

10.9.1 Compiling All Examples

You can compile and run the object behavior samples shipped with Solstice EM individually or as a group. Instructions for running each of the individual samples are provided in Section 10.9.2 “`cellSample`” on page 10-35” through Section 10.9.6 “`diskInfo`” on page 10-48.” In addition, Section 10.10 “Object Development Scenario Using Chai Object” on page 10-49” leads you through the entire process for the `chai` object in detail.

ODT provides a global Makefile that compiles all the object behavior examples. To build these examples, perform the following commands:

```
hostname# cd $EM_HOME/odt/src
hostname# Make all
```

Note – This mechanism does not currently compile the cellSample example. You must compile and run cellSample by itself.

10.9.2 cellSample

10.9.2.1 Important Code Functions

The cellSample example illustrates how to use the Object Services API. Specific sections of the code are not specifically identified as being more important than any others. You might want to look at all the code to see how the Object Services API can be used effectively.

To Build the Example

1. Go to the ODT examples directory.

```
hostname% cd $EM_HOME/src/odt/cellSample
```

2. Copy the cellSample GDMO and ASN.1 files to the appropriate directories.

```
hostname% cp cellSample.gdmo $EM_HOME/etc/gdmo
hostname% cp cellSample.asn1 $EM_HOME/etc/asn1
```

3. Load the GDMO into the MDR.

```
hostname% em_services -r
```

4. Generate the code for cellSample.

```
hostname% em_obcodegen cellSample
```

5. Create the dynamic linked library for the cellSample object class for default implementation.

```
hostname% make -f Makefile.cellSample extended
```

6. Load the cellSample source into an addressable location in the MIS.

```
hostname% ./cellSample.load
```

7. Restart the MIS.

```
hostname% $EM_HOME/bin/em_services
```

To Execute the Example

1. Create an instance of the cellSample object (instantiate the class).

```
./cellSample
```

2. Start OBED and run actions against the cellSample.

10.9.3 demoPing

10.9.3.1 Important Code Functions

The demoPing example shows how you can develop a simple native agent using the ODT.

Action Implementation

```
//-----//
//              ACTION IMPLEMENTATION              //
//-----//
// Switch for all actions specified in the GDMO definition of Managed //
// Object Class. Add the Action implementation in individual case //
// statements.//

// IMPORTANT NOTE:                                     //
// -----                                             //
// When implementing an Action, Please do not forget to return Action//
// result in cd.result for individual actions in switch statement.    //
//                                                                    //

// ODT_DEFAULT IMPLEMENTATION:                        //
// -----                                             //
// Default implementation returns a NULL Asn1Value and indicates //
// success by returning CHECK_DONE in CheckData.        //
//-----//

        switch(ai.local_value() )
        {
                case IDX_pingHost:

#ifdef ODT_EXTENDED
//***** $ODT_EXT_START [ACTION IMPLEMENTATION INSERT] *****//
        {

                DataUnit hostname;
                input.decode_octets(hostname);
                demoPing_error.print("IS %s\n",hostname.chp());

                if(!demoping(hostname, cb))
                {
                        cd.result = CheckData::CHECK_ERROR;
                        cb.exec(&cd);
                }
                return;
        }

//***** $ODT_EXT_END   [ACTION IMPLEMENTATION INSERT] *****//
#endif
        break;
```

```

        };
#ifdef ODT_DEFAULT
// Default implementation (returns NULL Action Response & Success)
    cd.result = CheckData::CHECK_DONE;
    cb.exec(&cd);
#endif
    }

    BEGHANDLERS
    CATCHALL {

#ifdef ODT_EXTENDED
//***** $ODT_EXT_START [ EXCEPTION HANDLING CATCHALL INSERT] *****//
//***** $ODT_EXT_END   [ EXCEPTION HANDLING CATCHALL INSERT] *****//
#endif

    }
    ENDHANDLERS
}

```

demoPing Callback Function

```
void
demoping_cb(Ptr userdata, Ptr calldata)

{
    CheckData cd;
    demoping_userdata *dl = (demoping_userdata *)userdata;
    DataUnit hostname = dl->hostname;

    struct sockaddr_in from;
    int len;
    char buf[1024];
    int fromlen=sizeof(from);

    if ( (len = recvfrom(dl->sockfd, (char *)buf, 1024 , 0,
        (sockaddr*)&from, &fromlen )) < 0)
    {
        demoPing_error.print("Failed in recvfrom() for ICMP Packet \n");
        cd.result = CheckData::CHECK_ERROR;
        dl->cb.exec(&cd);
        purge_fd_read_callback(dl->sockfd);
        close(dl->sockfd);
        delete dl;
        return;
    }

    demoPing_debug.print("received %d = %s\n",len,buf);
    demoPing_debug.print("from  %ld\n",from.sin_addr);

    pingreply_struct prpl;

    if(!pr_pack( (char *)buf, len, &prpl))
    {
        post_fd_read_callback(dl->sockfd,
            Callback((CallbackHandler)demoping_cb, dl));
        return;
    }

    close(dl->sockfd);
}
```

```

Asn1Value direply;
if(!make_pingrpl(&prpl,direply))
{
demoPing_error.print("Failed in encoding action reply\n");
cd.result = CheckData::CHECK_ERROR;
d1->cb.exec(&cd);
purge_fd_read_callback(d1->sockfd);
close(d1->sockfd);
delete d1;
return;
}
direply.print(demoPing_error);

cd.rv = direply;
cd.result = CheckData::CHECK_DONE;
purge_fd_read_callback(d1->sockfd);
close(d1->sockfd);
d1->cb.exec(&cd);
delete d1;
}

```

To Build the Example

1. Go to the ODT examples directory.

```
hostname% cd $EM_HOME/src/odt/demoPing
```

2. Copy the demoPing GDMO and ASN.1 files to the appropriate directories.

```

hostname% cp demoPing.gdmo $EM_HOME/etc/gdmo
hostname% cp demoPing.asn1 $EM_HOME/etc/asn1

```

3. Load the GDMO into the MDR.

```
hostname% em_services reload
```

4. Generate the code for demoPing.

```
hostname% em_obcodegen demoPing
```

5. Create the dynamic linked library for the demoPing object class for default implementation.

```
hostname% make -f Makefile.demoPing extended
```

6. Load the demoPing source into an addressable location in the MIS.

```
hostname% ./demoPing.load
```

7. Restart the MIS.

```
hostname% $EM_HOME/bin/em_services
```

To Execute the Example

1. Create an instance of the demoPing object (instantiate the class).

```
./pmi_demoPing
```

2. Start OBED and run action on instance specifying the hostname you want to ping.

3. Alternatively, you can run the ODT Sample Program driver (odtsamples) and select the Ping option.

10.9.4 demoregistry

10.9.4.1 Important Code Functions

Action Implementation

```
//-----//
//          ACTION IMPLEMENTATION          //
//-----//
// Switch for all actions specified in the GDMO definition of Managed //
// Object Class. Add the Action implementation in individual case //
// statements. //

// IMPORTANT NOTE: //
// ----- //
// When implementing an Action, Please do not forget to return Action//
// result in cd.result for individual actions in switch statement. //
// //

// ODT_DEFAULT IMPLEMENTATION: //
// ----- //
// Default implementation returns a NULL Asn1Value and indicates //
// success by returning CHECK_DONE in CheckData. //
//-----//

switch(ai.local_value() )
{
    case IDX_emDemoRegistryReg:
#ifdef ODT_EXTENDEED

//***** $ODT_EXT_START [ACTION IMPLEMENTATION INSERT] *****//
{
    Asn1Value hostasn1;
    Asn1Value appasn1;
    Asn1Value appidasn1;
    Asn1Value temp;
    DataUnit appname;
    DataUnit hostname;
    I32 appid;
```



```

        input.first_component(temp);
        temp.first_component(hostasn1);
        input.next_component(temp,temp);
        temp.first_component(appasn1);
        input.next_component(temp,temp);
        temp.first_component(appidasn1);

        hostasn1.decode_octets(hostname);
        appasn1.decode_octets(appname);
        appidasn1.decode_int(appid);

        if(!register_me(appname, hostname, appid, cb))
        {
            cd.result = CheckData::CHECK_ERROR;
            cb.exec(&cd);
        }
        return;
    }

//***** $ODT_EXT_END    [ACTION IMPLEMENTATION INSERT] *****//
#endif

        break;
        case IDX_emDemoRegistryValidateCookie:

#ifdef ODT_EXTENDED
//***** $ODT_EXT_START [ACTION IMPLEMENTATION INSERT] *****//
// Use it to implement some good cookie eating in your app
//***** $ODT_EXT_END    [ACTION IMPLEMENTATION INSERT] *****//
#endif

        break;
        case IDX_emDemoRegistryUnreg:

#ifdef ODT_EXTENDED
//***** $ODT_EXT_START [ACTION IMPLEMENTATION INSERT] *****//
{
    Asn1Value hostasn1;
    Asn1Value appasn1;
    Asn1Value appidasn1;
    Asn1Value temp;

    DataUnit appname;
    DataUnit hostname;
    I32 appid;

```

```

        input.first_component(temp);
        temp.first_component(hostasn1);
        input.next_component(temp,temp);
        temp.first_component(appasn1);
        input.next_component(temp,temp);
        temp.first_component(appidasn1);

        hostasn1.decode_octets(hostname);
        appasn1.decode_octets(appname);
        appidasn1.decode_int(appid);

        if(!unregister_me(appname, hostname, appid, cb))
        {
            cd.result = CheckData::CHECK_ERROR;
            cb.exec(&cd);
        }
        return;
    }

//***** $ODT_EXT_END    [ACTION IMPLEMENTATION INSERT] *****//
#endif

        break;
    };

#ifdef ODT_DEFAULT
    // Default implementation (returns NULL Action Response & Success)
    cd.result = CheckData::CHECK_DONE;
    cb.exec(&cd);
#endif

    }
    BEGHANDLERS
    CATCHALL {

#ifdef ODT_EXTENDED
//***** $ODT_EXT_START [ EXCEPTION HANDLING CATCHALL INSERT] *****//
//***** $ODT_EXT_END   [ EXCEPTION HANDLING CATCHALL INSERT] *****//
#endif

    }
    ENDHANDLERS
}

```

Function to Register Application

```
Result
register_me(DataUnit &appname, DataUnit &hostname, int appid,
           Callback &cb)
{
    demoregistry_userdata *dl=new demoregistry_userdata;
    emDemoRegister info;

    strcpy(info.appname,appname.chp());
    strcpy(info.hostname,hostname.chp());

    if(! demoClientSend( &info, DemoRegister, dl->sockfd,
hostname.chp() ) )
        return(NOT_OK);

    dl->cb = (Callback *)&cb;
    dl->appname = appname;
    dl->hostname = hostname;
    dl->appid = appid;

    post_fd_read_callback(dl->sockfd,
        Callback((CallbackHandler)demoregistry_cb, dl));

    return OK;
}
```

Function to Unregister Application

```
Result
unregister_me(DataUnit &appname, DataUnit &hostname, int appid,
             Callback &cb)
{
    demoregistry_userdata *dl=new demoregistry_userdata;
    emDemoRegister info;

    strcpy(info.appname,appname.chp());
    strcpy(info.hostname,hostname.chp());
```

```

        if(! demoClientSend( &info, DemoUnregister, dl->sockfd,
hostname.chp() ) )
            return(NOT_OK);

        dl->cb = (Callback *)&cb;
        dl->appname = appname;
        dl->hostname = hostname;
        dl->appid = appid;

        post_fd_read_callback(dl->sockfd,
            Callback((CallbackHandler)demounregistry_cb, dl));

        return OK;
    }

```

To Build the Example

1. Go to the ODT examples directory.

```
hostname% cd $EM_HOME/src/odt/demoregistry
```

2. Copy the demoregistry GDMO and ASN.1 files to the appropriate directories.

```
hostname% cp demoregistry.gdmo $EM_HOME/etc/gdmo
hostname% cp demoregistry.asn1 $EM_HOME/etc/asn1
```

3. Load the GDMO into the MDR.

```
hostname% em_services reload
```

4. Generate the code for demoregistry.

```
hostname% em_obcodegen demoregistry
```

5. Create the dynamic linked library for the demoregistry object class for default implementation.

```
hostname% make -f Makefile.demoregistry extended
```

6. Load the demoregistry source into an addressable location in the MIS.

```
hostname% ./demoregistry.load
```

7. Restart the MIS.

```
hostname% $EM_HOME/bin/em_services
```

10.9.4.2 Running the Example

1. Create an instance of demoregistry class (instantiate the class).

```
./pmi_demoregistry
```

2. Run demo_server on your local host or, if running on a remote host make sure Solstice EM is installed on the remote host.
3. Start OBED and find the object instance under EM-MIS.
4. Click on the OI and issue a DemoReg action where the ActionInfo parameter is {"hostname", "ApplicationName", 345}.
5. Alternatively, you can run the ODT Sample Program driver (odtsamples) and run the Register option or the UnRegister option.

10.9.5 demoServer

10.9.5.1 Building the Example

1. Go to the ODT examples directory.

```
hostname% cd $EM_HOME/src/odt/demoServer
```

2. Create the dynamic linked library for the demoServer object class for default implementation.

```
hostname% make
```

10.9.5.2 Running the Example

To start the demo_server:

```
> demo_server
```

10.9.6 diskInfo

10.9.6.1 Building the Example

1. Go to the ODT examples directory.

```
hostname% cd $EM_HOME/src/odt/diskInfo
```

2. Copy the diskInfo GDMO and ASN.1 files to the appropriate directories.

```
hostname% cp diskInfo.gdmo $EM_HOME/etc/gdmo  
hostname% cp diskInfo.asn1 $EM_HOME/etc/asn1
```

3. Load the GDMO into the MDR.

```
hostname% em_services reload
```

4. Generate the code for diskInfo.

```
hostname% em_obcodegen diskInfo
```

5. Create the dynamic linked library for the diskInfo object class for default implementation.

```
hostname% make -f Makefile.diskInfo extended
```

6. Load the diskInfo source into an addressable location in the MIS.

```
hostname% ./diskInfo.load
```

7. Restart the MIS.

```
hostname% $EM_HOME/bin/em_services
```

10.9.6.2 Running the Example

1. Create an instance of the `diskInfo` object (instantiate the class).

```
./pmi_diskInfo
```

2. Start OBED and run an action on the object instance, specifying the hostname about which you want to get disk information.
3. Alternatively, you can run the ODT Sample Program driver (`odtsamples`) and select the `DiskInfo` of a Host option.

10.10 Object Development Scenario Using Chai Object

The sample files shipped with Solstice EM for object development scenarios provide important information in README files. Although the information in the README files is fairly complete, this section of the documentation provides an expanded view of the object development scenario for the Chai managed object.

10.10.1 Creating Your Own Object Class

1. Load the chai managed object into the Meta Data Repository (MDR).

```
# em_gdmo hostname chai.gdmo
# em_asnl -o 'pwd' chai.asnl
# cp 1.3.6.1.4.1.42.2.2.2.1.96.3.1
/var/opt/SUNWconn/em/usr/data/ASN1
# cp *-ASN1 /var/opt/SUNWconn/em/usr/data/ASN1
```

2. Define environment variables, if needed.

Location of hidden/intermediate files:

HIDDENDIR=/tmp

Data storage for OC whether PERSISTENT or VOLATILE:

DATASTORAGE=PERSISTENT

3. Go to the directory where you want your code to be generated.

```
% cd user_directory
```

4. Generate the C++ code for your objects.

You will see output similar to the following:

```
hostname% em_obcodegen chai
objdefn_info:  Attribute Name is chaiKettleNumber
objdefn_info:  Attribute Name is chaiBlend
objdefn_info:  Attribute Name is chaiReady
objdefn_info:  Attribute Name is discriminatorConstruct
objdefn_info:  Attribute Name is administrativeState
objdefn_info:  Attribute Name is operationalState
objdefn_info:  Total Number Of Attributes is 6
objdefn_info:  *****
objdefn_info:  Action Name is brewChai
objdefn_info:  Total Number Of Actions is 1
objdefn_info:  *****
objdefn_info:  Name Binding Name is chai-system
objdefn_info:  Name Binding Name is chai-chai
objdefn_info:  Total Number Of Name Bindings is 2
objdefn_info:  *****
```

Note that OCG identifies the attributes, actions, and name bindings for which code will be generated.

The following files are generated:

- Makefile.chai
- README.chai
- chai.load
- chai.unload
- chai_user.odt.cc
- chai_user.odt.hh
- pmi_chai.cc

5. Compile and make the dynamic library for the default implementation.

```
% make -f Makefile.chai default
```

This command compiles and makes a dynamic library called `chai.so`.

6. Create a customized implementation.

To create a customized implementation, you need to first modify the source code. Then, to compile and make a customized library, use the following format:

```
% make -f Makefile.chai extended
```

7. Load the new dynamic library.

```
% chai.load
```

8. Terminate and restart the MIS.

```
# em_services
```

9. Instantiate the new chai object.

```
% cd chai_Create_program  
% make chai  
% chai// This creates the chai object  
% chai -g// This gets attributes of new chai object
```

10. Run the debugger to verify the object behaves as expected.

```
% debugger $EM_HOME/bin/em_mis &  
    stop in DynLoader::DynLoader  
    run
```

10.10.2 Debugging Flags

The following code lines that contain the debug agents for the chai object class are included in the `chai_user.odt.cc` file.

```
Debug_on(cahi_info)
Debug_on(chai_error)
```

If debug agents are spread across multiple files, the above definitions must only be included in the `chai_user.hh` file (copied and modified from `chai_user.odt.hh`). The other files need to contain the following code lines:

```
extern Debug chai_info;
extern Debug chai_error;
```

10.10.3 Sample Behavior Implementation

The following program implements specialized behavior for the `chaiReady` attribute. If `chaiReady` is 0, then set the `chaiBlend` to “Earl Grey” and send `brewAction`.

To add this behavior, insert the following piece of code in `chai_user.cc` at the location of `chai_AttrSecty::read` after the subread is performed.

```
// User Behavior Extension: Start
// Def: If the chaiReady is equal to 0 then
//      Set the blend to “Earl Grey” and then send a action
//      to brewchai.

// SectyInfo definition
AttrSectyInfo    AttrInfo;
ActionSectyInfo  ActionInfo;

Asn1Value  ready, blend, Orig_Blend;
I32  Is_chai_Ready;

// Get original index of our Attribute
int org_index = AttrSectyInfo2index(ai);
```

```

// Check to see we're reading chaiReady, if yes, then specialize
// the behavior
    if (org_index == IDX_chaiReady) {
        if (av) {
            av.decode_int(Is_chai_Ready);
            if (!Is_chai_Ready)
            {
                // We're out of chai.. brew and choose my blend
                // Earl Grey
                index2AttrSectyInfo(IDX_chaiBlend, AttrInfo);
                (void) fetch(AttrInfo, NULL_CALLBACK);
                (void) read(AttrInfo, Orig_Blend);

                DataUnit O_blend;
                Asn1Value new_blend;

                Orig_Blend.decode_octets(O_blend);
                chai_debug.print("Read Secretary - chai_Blend is\n");
                Orig_Blend.print(chai_debug);
                if (O_blend != DataUnit("Earl Grey"))
                {
                    // Special check to blend only Earl Grey
                    DataUnit chai_blend("Earl Grey");

                    new_blend.encode_octets(TAG_OCTSTR, chai_blend);
                    (void) write(AttrInfo, new_blend);

                    (void) store(AttrInfo, NULL_CALLBACK);
                }
                // Go Ahead and Brewchai
                index2ActionSectyInfo(IDX_brewchai, ActionInfo);
                action(ActionInfo, new_blend, NULL_CALLBACK);
            }
        }
    }
}

```

10.10.4 chai Object Class Definitions

The example GDMO and ASN.1 definitions for the chai object class are installed into the `$EM_HOME/src/odt/chai` directory when you install the ODT onto your system (SUNWemobj package).

The `chai.gdmo` file defines the following attributes:

- `chaiKettleNumber`
- `chaiBlend`
- `chaiReady`

The `chai.gdmo` file also defines the `brewChai` action.

10.10.4.1 Sample chai.gdmo Definitions File

```
-- Copyright 03 Apr 1996 Sun Microsystems, Inc. All Rights Reserved.--
-- #pragma ident  "@(#)chai.gdmo1.2 96/04/03 Sun Microsystems"

MODULE "EM Chai Document"

basechai MANAGED OBJECT CLASS
    DERIVED FROM "Rec. X.721 | ISO/IEC 10165-2 : 1992" : top;

    CHARACTERIZED BY
        chaiPackage;
    REGISTERED AS { em-chai-objectClass 0 };

chaiPackage PACKAGE
    BEHAVIOUR chaiPackageDefinition BEHAVIOUR DEFINED AS
        !This managed object class represents the chai
        from the neighbourhood chai shop !;
    ;
    ATTRIBUTES
        chaiKettleNumber GET-REPLACE,
        chaiBlend    GET-REPLACE,
        chaiReady    GET-REPLACE,
        "Rec. X.721 | ISO/IEC 10165-2 : 1992" : discriminatorConstruct
            REPLACE-WITH-DEFAULT
            DEFAULT VALUE Attribute

ASN1Module.defaultDiscriminatorConstruct
    GET-REPLACE,
    "Rec. X.721 | ISO/IEC 10165-2 : 1992" : administrativeState
        GET-REPLACE,
    "Rec. X.721 | ISO/IEC 10165-2 : 1992" : operationalState
        GET;
    ACTIONS
        brewChai;
    NOTIFICATIONS
        "Rec. X.721 | ISO/IEC 10165-2 : 1992" :
            objectCreation,
        "Rec. X.721 | ISO/IEC 10165-2 : 1992" :
            objectDeletion,
        "Rec. X.721 | ISO/IEC 10165-2 : 1992" :
            attributeValueChange;
    REGISTERED AS { em-chai-package 1 };
```

```

chai MANAGED OBJECT CLASS
    DERIVED FROM basechai;
    REGISTERED AS { em-chai-objectClass 1 };

-- Actions

brewChai ACTION
    MODE CONFIRMED;
    WITH INFORMATION SYNTAX Chai-ASN1.ChaiString;
    WITH REPLY SYNTAX Chai-ASN1.ChaiString;
    REGISTERED AS { em-chai-action 1 };

-- Name Bindings

chai-system NAME BINDING
    SUBORDINATE OBJECT CLASS chai;
    NAMED BY
    SUPERIOR OBJECT CLASS "Rec. X.721 | ISO/IEC 10165-2 : 1992" : system;
    WITH ATTRIBUTE chaiKettleNumber;
    BEHAVIOUR chai-rootBehaviour BEHAVIOUR DEFINED AS
    !This name is used to define the chai object
    name binding!;
    ;
    CREATE;
    DELETE ONLY-IF-NO-CONTAINED-OBJECTS;
    REGISTERED AS { em-chai-binding 1 };

chai-chai NAME BINDING
    SUBORDINATE OBJECT CLASS chai;
    NAMED BY
    SUPERIOR OBJECT CLASS chai;
    WITH ATTRIBUTE chaiKettleNumber;
    BEHAVIOUR chai-systemchai BEHAVIOUR DEFINED AS
    !This name is used to define the chai object
    name binding under system branch!;
    ;
    CREATE;
    DELETE ONLY-IF-NO-CONTAINED-OBJECTS;
    REGISTERED AS { em-chai-binding 2 };

-- Attributes

```

```
chaiKettleNumber ATTRIBUTE
    WITH ATTRIBUTE SYNTAX Chai-ASN1.ChaiInteger;
    MATCHES FOR EQUALITY;
    BEHAVIOUR chaiKettleNumberBehaviour BEHAVIOUR DEFINED AS
    !This is the naming attribute for the chai
    object.!!;
    ;
    REGISTERED AS { em-chai-attribute 1 };

chaiBlend ATTRIBUTE
    WITH ATTRIBUTE SYNTAX Chai-ASN1.ChaiString;
    MATCHES FOR EQUALITY;
    BEHAVIOUR chaiBlendBehaviour BEHAVIOUR DEFINED AS
    !This is the blend of chai that is brewing
    in the current Kettle!!;
    ;
    REGISTERED AS { em-chai-attribute 2 };

chaiReady ATTRIBUTE
    WITH ATTRIBUTE SYNTAX Chai-ASN1.ChaiInteger;
    MATCHES FOR EQUALITY;
    BEHAVIOUR chaiReadyBehaviour BEHAVIOUR DEFINED AS
    !If this attribute is true there is chai in
    the Kettle!!;
    ;
    REGISTERED AS { em-chai-attribute 3 };

END
```

10.10.4.2 Sample chai.asn1 Definitions File

```
-- Copyright 03 Apr 1996 Sun Microsystems, Inc. All Rights Reserved.--
-- #pragma ident  "@(#)chai.asn1.2 96/04/03 Sun Microsystems

Chai-ASN1
{iso(1) org(3) dod(6) internet(1) private(4) enterprises(1) sun(42)
  products(2) management(2) em(2) odt(1) em-chai(96)
asn1Module(2) 0}

DEFINITIONS ::=
BEGIN
em-chai OBJECT IDENTIFIER ::=
  {iso(1) org(3) dod(6) internet(1) private(4) enterprises(1) sun(42)
    products(2) management(2) em(2) odt(1) em-chai(96)}

em-chai-objectClass  OBJECT IDENTIFIER ::= { em-chai 3 }
em-chai-package      OBJECT IDENTIFIER ::= { em-chai 4 }
em-chai-binding      OBJECT IDENTIFIER ::= { em-chai 6 }
em-chai-attribute    OBJECT IDENTIFIER ::= { em-chai 7 }
em-chai-action       OBJECT IDENTIFIER ::= { em-chai 9 }

ChaiInteger ::= INTEGER
ChaiString  ::= GraphicString
ChaiBoolean ::= BOOLEAN
END
```

10.10.5 Sample PMI Program to Create a New chai Object Instance

```
/*
"This file is generated using Solstice EM (2.0) - Object Development Tools" Code
Generator
*/
```



```

#include <hi.hh>
#include <error.hh>
#include <sys/types.h>
#include <unistd.h>
#ifdef HPUX
#include <sys/param.h>
#else
#include <sys/systeminfo.h>
#endif

void create_chai( DU &dn);

main(int argc, char **argv)
{
    printf("MODIFY THE GENERATED CODE FILE ./pmi_chai.cc \n");
    exit(0);

    Platform plat(duEM);

    if (plat.get_error_type() != PMI_SUCCESS)
    {
        printf("Platform constructor failed...\n");
        printf("Reason: %s\n", plat.get_error_string());
        exit(1);
    }

    // Initialize to the dn of object

    // dn can be a name starting from local root or fully distinguished name
    DU dn; /* DISTINIGUISHED NAME OF OBJECT HERE*/

    // Connect to the mis running on the local host
    if (!plat.connect("localhost", "chai_sample"))
    {
        printf("Connecting to platform Failed \n");
        printf("Reason: %s\n", plat.get_error_string());
        exit(2);
    }

    // Create the object
    create_chai(dn);
}

```

```

void
display_attributes(Image &im)
{

    // Get all the attribute names in an Array of DataUnits.
    // Perform a get on each attribute to get its value
    // Note we have stripped off the document name to make
    // the attribute value pairs more readable
    // the chp() method of the DataUnit is necessary to null
    // terminate the DataUnit.
    //

    Array(DU) attr_names = im.get_attr_names();
    fprintf(stdout, "Attribute\tValue\n-----\t-----\n");
    for (int i=0; i<attr_names.size; i++) {
        DU& name = attr_names[i];
        // note: next to lines use chp() function to convert a
        // DataUnit into char *.
        char *short_name = strrchr(attr_names[i].chp(), ':');
        fprintf(stdout, "%s: \t%s \n", ++short_name,
            im.get_str(name, USE_EXPLICIT_CHOICE|OMIT_NEWLINES).chp());
    }
    fprintf(stdout, "\n\n");
    fflush(stdout);
}

void
create_chai(DU &dn)
{

    Image im;
    im = Image(dn, DU("chai"));

    if (!im.boot())
    {
        printf("Image::boot Failed %s\n", im.get_error_string());
        exit(3);
    }
}

```

```

/* UNCOMMENT AND MODIFY THE APPROPRIATE LINES IF YOU WANT TO CREATE
 * OBJECT WITH SOME ATTRIBUTE VALUES
if (!im.set_str("chaiKettleNumber", "None"))
{
printf("Image::set_str() failed for chaiKettleNumber: %s %d\n",
      im.get_error_string(), im.get_error_type());
exit(4);
}
if (!im.set_str("chaiBlend", "None"))
{
printf("Image::set_str() failed for chaiBlend: %s %d\n",
      im.get_error_string(), im.get_error_type());
exit(4);
}
if (!im.set_str("chaiReady", "None"))
{
printf("Image::set_str() failed for chaiReady: %s %d\n",
      im.get_error_string(), im.get_error_type());
exit(4);
}
*
*/
if (im.get_error_type() != PMI_SUCCESS)
{
printf("Image::set_str Failed %s\n",im.get_error_string());
exit(5);
}

if (!im.create())
{
printf("Create Failed %s\n",im.get_error_string());
exit(6);
}
printf("Created instance %s\n", dn.chp());
display_attributes(im);
}

```

10.10.6 Example Generated Code in .cc File

The following generated code stub examples are based on the `chai` example object. A complete scenario for defining the `chai` object is provided in Section 10.10 “Object Development Scenario Using Chai Object” on page 10-49.” The actual code that OCG generates can contain additional comments not reflected in this book.

Note – Throughout these examples and in any code generated by OCG, there are lines that are similar to the following:

```
//***** $ODT_EXT_START [LOCAL VARIABLE INSERT] *****//  
//***** $ODT_EXT_END [LOCAL VARIABLE INSERT] *****//
```

These lines indicate areas in the code where you can safely add your own code to the generated code to further customize object behaviors.

10.10.6.1 Generated Asynchronous Read Stub Function (FETCH)

Function

```
chai_AttrSecty::fetch (Const AttrSectyInfo &ai, Const callback  
                      &cb)
```

Description

This function is an asynchronous interface for reading attributes. For every attribute requested in a `GET` or `SET` request, the MIS Framework calls `fetch` for that attribute, followed by a `read` of the same attribute. The `fetch` function can reflect status back to the invoker by:

- Using the return value from the function
- Throwing an MIS exception
- Passing a parameter to the invocation of the callback

Arguments

This function uses the following arguments:

- *ai* indicates the attribute to be read
- *cb* identifies the callback routine passed by Framework

Return Value

This function returns the following values:

- OK if the attribute is fetched and read successfully
- NOT_OK to indicate failure in fetching/reading attribute specified by `ai`

Errors

- NOT_OK

SET_LIST_ERROR or GET_LIST_ERROR

- MIS Exceptions

Any exception results in an SET_LIST_ERROR or a GET_LIST_ERROR. **If an exception is thrown the callback must not be invoked**

- Operr(ErrorMessage)
- Operr(intval)
- Operr(probableCauseOid);
- Operr(errorId, errorValue)

The Operr data structure is passed to the Object Framework by invoking the passed `exec` with the parameter set to a pointer to the allocated Operr data structure. The default behavior is for the subordinate secretary to invoke the callback.

Note – A NULL parameter to the invocation of the callback indicates a POSITIVE status e.g `cb.exec((Ptr) 0)`; If the ODT developer wishes to return an error, an Operr Data structure must be allocated from the heap and passed to the callback function.

Code Example

```
Result
chai_AttrSecty::fetch(const AttrSectyInfo &ai, const Callback &cb)
{
    TRACE(Tracer TR(chai_trace, "chai_AttrSecty::fetch",
        "this = 0x%lx, const AttrSectyInfo &ai = 0x%lx, const Callback &cb = "
        "0x%lx", (void*)this, &ai, (void*)cb));

#ifdef ODT_EXTENDED
    /******* $ODT_EXT_START [LOCAL VARIABLE INSERT] *****/
    /******* $ODT_EXT_END   [LOCAL VARIABLE INSERT] *****/
#endif

    TRY
    {
        // Fetch attribute specified by (ai)
        return subfetch(ai,cb);

#ifdef ODT_EXTENDED
    /****** $ODT_EXT_START [FETCH BEHAVIOUR SPECIALIZATION INSERT] **//
    /****** $ODT_EXT_END   [FETCH BEHAVIOUR SPECIALIZATION INSERT] **//
#endif

    }
    BEGHANDLERS
    CATCHALL {

#ifdef ODT_DEFAULT
        return (NOT_OK);
#endif

#ifdef ODT_EXTENDED
    /****** $ODT_EXT_START [ EXCEPTION HANDLING CATCHALL INSERT] *****/
    /****** $ODT_EXT_END   [ EXCEPTION HANDLING CATCHALL INSERT] *****/
#endif

    }
    ENDHANDLERS
}
```

10.10.6.2 Generated Asynchronous Write Stub Function (STORE)

Function

```
chai_AttrSecty::store (Const AttrSectyInfo &ai, Const callback  
                      &cb)
```

Description

This function is an asynchronous interface for storing attributes. For every attribute requested in a SET request, MIS Framework calls `write` for that attribute, followed by a `store` of the same attribute. The store function can reflect status back to the invoker by:

- using the return value from the function
- throwing an MIS exception
- passing a parameter to the invocation of the callback

Arguments

This function uses the following arguments:

- *ai* indicates the attribute to be stored
- *cb* identifies the callback routine passed by Framework

Return Value

This function returns the following values:

- OK if the attribute is written and stored successfully
- NOT_OK to indicate failure in writing/storing attribute specified by *ai*

Errors

- NOT_OK

SET_LIST_ERROR

- MIS Exceptions

Any exception results in a Process Failure **If an exception is thrown the callback must not be invoked**

- Operr(ErrorMessage)

- `Operr(intval)`
- `Operr(probableCauseOid);`
- `Operr(errorId, errorValue)`

The `Operr` data structure is passed to the Object Framework by invoking the passed `exec` with the parameter set to a pointer to the allocated `Operr` data structure. The default behavior is for the subordinate secretary to invoke the callback.

Note – A `NULL` parameter to the invocation of the callback indicates a `POSITIVE` status e.g `cb.exec((Ptr) 0);` If the ODT developer wishes to return an error, an `Operr` Data structure must be allocated from the heap and passed to the callback function.

Code Example

```
Result
chai_AttrSecty::store(const AttrSectyInfo &ai, const Callback &cb)
{
    TRACE(Tracer TR(chai_trace, "chai_AttrSecty::store",
        "this = 0x%lx, const AttrSectyInfo &ai = 0x%lx, const Callback &cb = "
        "0x%lx", (void*)this, &ai, (void*)cb));

#ifdef ODT_EXTENDED
    //***** $ODT_EXT_START [LOCAL VARIABLE INSERT] *****//
    //***** $ODT_EXT_END   [LOCAL VARIABLE INSERT] *****//
#endif

    TRY
    {
        // store attribute to DATASTORAGE specified by (ai)
        return substore(ai,cb);

#ifdef ODT_EXTENDED
        //***** $ODT_EXT_START [STORE BEHAVIOUR SPECIALIZATION INSERT] **//
        //***** $ODT_EXT_END   [STORE BEHAVIOUR SPECIALIZATION INSERT] **//
#endif

    }
    BEGHANDLERS
    CATCHALL {

#ifdef ODT_DEFAULT
        return (NOT_OK);
#endif

#ifdef ODT_EXTENDED
        //***** $ODT_EXT_START [ EXCEPTION HANDLING CATCHALL INSERT] *****//
        //***** $ODT_EXT_END   [ EXCEPTION HANDLING CATCHALL INSERT] *****//
#endif

    }
    ENDHANDLERS
}
```

10.10.6.3 Generated Synchronous Read Stub Function (READ)

Function

```
chai_AttrSecty::read (Const AttrSectyInfo &ai, Asn1Value &av)
```

Description

This function is a synchronous interface for reading attributes. For every attribute requested in a GET or SET request, MIS Framework calls `read` for that attribute. The read function can reflect status back to the invoker by:

- using the return value from the function
- throwing an MIS exception

Arguments

This function uses the following arguments:

- *ai* indicates the attribute to be read
- *av* identifies the Asn1Value of the attribute read (output parameter)

Return Value

This function returns the following values:

- OK if the attribute is read successfully
- NOT_OK to indicate failure

Errors

- NOT_OK
 - SET_LIST or GET_LIST Error
- MIS Exceptions

All exceptions result in a SET_LIST or GET_LIST Error except in the case of a ResourceLimit exception which generates a ResourceLimit Error.

Code Example

```
Result
chai_AttrSecty::read(const AttrSectyInfo &ai, Asn1Value &av)
{
    TRACE(Tracer TR(chai_trace, "chai_AttrSecty::read",
        "this = 0x%lx, const AttrSectyInfo &ai = 0x%lx, Asn1Value &av = "
        "0x%lx", (void*)this, &ai, (void*)av));

#ifdef ODT_EXTENDED
//***** $ODT_EXT_START [LOCAL VARIABLE INSERT] *****/
//***** $ODT_EXT_END   [LOCAL VARIABLE INSERT] *****/
#endif

    TRY
    {
        int index = AttrSectyInfo2index(ai);

        // Validate passed attribute index
        if(!is_valid_index(index))
        {
            return NOT_OK;
        }

        // Read in-memory value of attribute specified by (ai)
        TRYRES (subread(ai,av));

        // Assign read attribute value
        index2lhsvalue(index) = av ;

#ifdef ODT_EXTENDED
//***** $ODT_EXT_START [READ BEHAVIOUR SPECIALIZATION INSERT] *****/
//***** $ODT_EXT_END   [READ BEHAVIOUR SPECIALIZATION INSERT] *****/
#endif
        return(OK);
    }
    BEGHANDLERS
    CATCHALL {

#ifdef ODT_DEFAULT
        return (NOT_OK);
#endif
    }
}
```

```

#ifdef ODT_EXTENDED
//***** $ODT_EXT_START [ EXCEPTION HANDLING CATCHALL INSERT] *****//
//***** $ODT_EXT_END   [ EXCEPTION HANDLING CATCHALL INSERT] *****//
#endif

    }
    ENDBHANDLERS
}

```

10.10.6.4 Generated Synchronous Write Stub Function (WRITE)

Function

```

chai_AttrSecty::write (Const AttrSectyInfo &ai, Const Asn1Value
    &av)

```

Description

This function is a synchronous interface for writing attributes. For every attribute requested in a SET request, MIS Framework calls `write` for that attribute. The write function can reflect status back to the invoker by:

- using the return value from the function
- throwing an MIS exception

Arguments

This function uses the following arguments:

- *ai* indicates the attribute to be written
- *av* identifies the `Asn1Value` of the attribute written (output parameter)

Return Value

This function returns the following values:

- OK if the attribute is written successfully
- NOT_OK to indicate failure

Errors

■ NOT_OK

SET_LIST or GET_LIST Error

■ MIS Exceptions

All exceptions result in a ProcessFailure.

Code Example

```
Result
chai_AttrSecty::write(const AttrSectyInfo &ai,
                      const Asn1Value &av)
{
    TRACE(Tracer TR(chai_trace, "chai_AttrSecty::write",
        "this = 0x%lx, const AttrSectyInfo &ai = 0x%lx, const Asn1Value "
        "&av = 0x%lx", (void*)this, &ai, (void*)av));

#ifdef ODT_EXTENDED
//***** $ODT_EXT_START [LOCAL VARIABLE INSERT] *****/
//***** $ODT_EXT_END   [LOCAL VARIABLE INSERT] *****/
#endif

    TRY
    {
        int index = AttrSectyInfo2index(ai);

        // Validate passed attribute index
        if(!is_valid_index(index))
        {
            return NOT_OK;
        }

        // Assign written attribute value
        index2lhsvalue(index) = av ;

        // Execute Write thru
        TRYRES (subwrite(ai,av));

#ifdef ODT_EXTENDED
//***** $ODT_EXT_START [WRITE BEHAVIOUR SPECIALIZATION INSERT ] **//
//***** $ODT_EXT_END   [WRITE BEHAVIOUR SPECIALIZATION INSERT ] **//
#endif
    }
}
```

```

        return(OK);
    }

    BEGHANDLERS
    CATCHALL {

#ifdef ODT_DEFAULT
        return (NOT_OK);
#endif

#ifdef ODT_EXTENDED
//***** $ODT_EXT_START [ EXCEPTION HANDLING CATCHALL INSERT] *****//
//***** $ODT_EXT_END   [ EXCEPTION HANDLING CATCHALL INSERT] *****//
#endif

    }
    ENDHANDLERS
}

```

10.10.6.5 Generated Action Stub Function (ACTION)

Function

```
chai_AttrSecty::action (Const AttrSectyInfo &ai, Const AsnlValue
                        &input, Const callback &cb)
```

Description

This function is an asynchronous interface for handling CMIP ACTIONS. The CMIP actions handled are those specified in the Managed Object Class definition in the GDMO. The action stub function can issue status to the invoker using 2 methods:

- Throwing an MIS Exception
- As a Parameter to the Callback

Arguments

This function uses the following arguments:

- *ai* indicates the action
- *input* indicates the Asn1Value specified by the user through the INFORMATION SYNTAX clause in the GDMO ACTION definition
- *cb* identifies the callback routine passed by Framework to process *cb.exec* (CheckData) where CheckData contains the returned ACTION RESPONSE and ACTION RESULT

Return Value

This function returns no values aside from those passed back in the *cb* argument. CheckData return values are:

- If Operation completed successfully
 - Set *cd.result* to CHECK_DONE.
 - Set *cd.rv* to encoded Asn1Value of ACTION RESPONSE.
- If Operation terminated with an error the developer has 2 options
 - Throw an MIS exception which will result in a ProcessingFailure. The callback need not be invoked
 - Use the Checkdata Structure to indicate the error. Set *cd.result* to CHECK_ERROR and fill in *cd.error* with an Error Response Message. If *cd.error* is not set the framework will issue a processing failure error.

Sample Error Generating Code

The following code illustrates how to build an Action Response message. The `action_type` Oid is found in the `ai` and needs to be encoded using `CONTEXT(2)` as the tag. The object instance and object class information can be found from the `moi` reference. The `action_reply` is an any defined by the `action_type`. This needs to be encoded according to the syntax of the `action_reply_syntax`:

```
// Encode the action Type using Oid from the ai parameter
Asn1Value action_type;
action_type.encode_oid(TAG_CONT(2), ai.id);

// allocate and fill in Error Resp, need
// to complete the OI, OC, action_Type and action reply fields
ActionRes *resp;
if ( !(resp = (ActionRes *)Message::new_message(ACTION_RES)) ||
    !(resp->action_type = action_type) ||
    resp->oc.encode_oid(TAG_CONT(0), moi->object_class()) !=OK ||
    !(resp->oi = moi->fdn())) {
    THROW(ResourceLimX);
}
Asn1Value action_reply;
// Here we need to encode the action reply, consider a reply syntax
// of
// INTEGER ::= {
//   noerror(0),
//   nobandwidth(1)
// }
// To encode a nobandwidth error
action_reply.encode_int(TAG_INT,1);
resp->action_reply = action_reply;
cd.error = (ResMess *) resp;
cd.result = CHECK_ERROR;
cb.exec(&cd);
```

The same method above could be used to create any response message defined in `message.hh` that would make sense for this action request.

Code Example

```
void
chai_ActionSecty::action(const ActionSectyInfo &ai,
                        const Asn1Value &input,
                        const Callback &cb)
{
    TRACE(Tracer TR(chai_trace, "chai_ActionSecty::action",
        "this = 0x%lx, const ActionSectyInfo &ai = 0x%lx, const Asn1Value "
        "&input = 0x%lx, const Callback &cb = 0x%lx",
        (void*)this, &ai, (void*)input, (void*)cb ));

#ifdef ODT_EXTENDED
//***** $ODT_EXT_START [LOCAL VARIABLE INSERT] *****//
//***** $ODT_EXT_END   [LOCAL VARIABLE INSERT] *****//
#endif

    TRY
    {

        CheckData cd;                                // CheckData (cd) - Action Response value //

        switch(ai.local_value() )
        {
            case IDX_brewChai:
#ifdef ODT_EXTENDED
//***** $ODT_EXT_START [ACTION IMPLEMENTATION INSERT] *****//
//***** $ODT_EXT_END   [ACTION IMPLEMENTATION INSERT] *****//
#endif
                break;

        };

#ifdef ODT_DEFAULT
// Default implementation (returns NULL Action Response & Success)
    cd.result = CheckData::CHECK_DONE;
    cb.exec(&cd);
#endif

    }
}
```

```

    BEGHANDLERS
    CATCHALL {

#ifdef ODT_EXTENDED
//***** $ODT_EXT_START [ EXCEPTION HANDLING CATCHALL INSERT] *****/
//***** $ODT_EXT_END   [ EXCEPTION HANDLING CATCHALL INSERT] *****/
#endif

    }
    ENDHANDLERS
}

```

10.10.6.6 Generated Instance Create Stub Function (CREATE)

Function

```

chai_AttrSecty::create_vote (Const Asn1Value &fdn,
                             Const Asn1Value &av)

```

Description

This function lets you validate a CMIP CREATE request before the infrastructure creates the object instance. The function is passed a resolved attribute list and FDN, which you can validate before voting OK or NOT_OK.

Arguments

This function uses the following arguments:

- *fdn* is the FDN (Fully-Distinguished Name) of the object instance to be created.
- *av* identifies the Asn1Value of the resolved Attribute List.

Return Value

This function returns the following values:

- OK to go ahead and create the object instance.
- NOT_OK to not create the object instance.

Code Example

```
Result
chai_InstanceSecty::create_vote(const Asn1Value &fdn,
                                const Asn1Value &av)
{
    TRACE(Tracer TR(chai_trace, "chai_InstanceSecty::create",
        "this = 0x%lx, const Asn1Value &fdn = 0x%lx, const Asn1Value "
        "&av = 0x%lx", (void*)this, &ai, (void*)fdn, (void*)av ));

#ifdef ODT_EXTENDED
//***** $ODT_EXT_START [LOCAL VARIABLE INSERT] *****//
//***** $ODT_EXT_END   [LOCAL VARIABLE INSERT] *****//
#endif

    TRY
    {

#ifdef ODT_EXTENDED
//***** $ODT_EXT_START [CREATE VOTE SPECIALIZATION INSERT ] *****//
//***** $ODT_EXT_END   [CREATE VOTE SPECIALIZATION INSERT ] *****//
#endif

        return OK;
    }

    BEGHANDLERS
    CATCHALL {

#ifdef ODT_DEFAULT
        return (NOT_OK);
#endif
    }

#ifdef ODT_EXTENDED
//***** $ODT_EXT_START [ EXCEPTION HANDLING CATCHALL INSERT] *****//
//***** $ODT_EXT_END   [ EXCEPTION HANDLING CATCHALL INSERT] *****//
#endif

    }
    ENDHANDLERS
}
```

10.10.6.7 Generated Instance Destroy Stub Function (DELETE)

Function

```
chai_AttrSecty::destroy_vote()
```

Description

This function lets you validate a CMIP DELETE request before the infrastructure deletes the object instance.

Arguments

This function uses no arguments.

Return Value

This function returns the following values:

- OK to delete the object instance.
- NOT_OK to not delete the object instance.

Code Example

```
Result
chai_AttrSecty::destroy_vote()
{
    TRACE(Tracer TR(chai_trace, "chai_InstanceSecty::destroy_vote",
        "this = 0x%lx", (void*)this));

#ifdef ODT_EXTENDED
//***** $ODT_EXT_START [LOCAL VARIABLE INSERT] *****//
//***** $ODT_EXT_END   [LOCAL VARIABLE INSERT] *****//
#endif

    TRY
    {
```

```

#ifdef ODT_EXTENDED
//***** $ODT_EXT_START [DELETE VOTE SPECIALIZATION INSERT ] *****//
//***** $ODT_EXT_END   [DELETE VOTE SPECIALIZATION INSERT ] *****//
#endif

    return OK;
}

    BEGHANDLERS
    CATCHALL {

#ifdef ODT_DEFAULT
        return (NOT_OK);
#endif

#ifdef ODT_EXTENDED
//***** $ODT_EXT_START [ EXCEPTION HANDLING CATCHALL INSERT] *****//
//***** $ODT_EXT_END   [ EXCEPTION HANDLING CATCHALL INSERT] *****//
#endif

    }
    ENDHANDLERS
}

```

10.10.6.8 Generated Receive Event Stub Function (RECEIVE_EVENT)

Function

```
chai_AttrSecty::receive_event (Asn1Value &event_type,  
                               Asn1Value &event_info)
```

Description

This function lets you receive events and notifications specified in `event_type` and `event_info`. You would provide specialized processing depending on `event_type` and `event_info` which have been received as the discriminatorConstruct specified in the object instance.

Note – The `receive_event` function only allow the user to access the `event_type` and `event_info`. This is a known problem. To access the complete event message place code in the `invoke_moi` function in the `ObjectClassimpl.cc` file which lives in the `hidden` directory. In the `cmd == discriminator_match` section the developer can access the complete event message by casting `parm` to be an `EventReq *`. See example below.

```
Result chai_MOI::invoke_moi( void * s, const Command &cmd, void *parm )  
{  
    if (cmd == discriminator_match)  
    {  
        // INSERT YOUR CODE HERE and cast parm to be an  
        // and EventReq Message.  
        Eventreq *p_event = (EventReq *) parm;  
        // Use p_event to access oi,oc,event_time,event_type  
        // and event_info  
        // Decide if receive_event needs to be Called  
        TRYRES(attrsecty->receive_event(  
            ((EventReq *)parm)->event_type,  
            ((EventReq *)parm)->event_info) );  
        return OK;  
    }  
    return NOT_OK;  
}
```

Note – This function is generated only if you specify the three discriminator attributes (DiscriminatorConstruct, OperationalState, and AdministrativeState) in your GDMO file and specify FILTER_ATTR: DiscriminatorConstruct in your configuration file.

Arguments

This function uses the following arguments:

- *event_type* is the type of event or notification received
- *event_info* is the Event Info received

Return Value

This function returns the following values:

- OK if the event or notification was processed successfully
- NOT_OK to indicate failure in processing event or notification

Code Example

```
Result chai_AttrSecty::receive_event(  
    Asn1Value &event_type, Asn1Value &event_info)  
{  
  
#ifdef ODT_EXTENDED  
//***** $ODT_EXT_START [LOCAL VARIABLE INSERT] *****//  
//***** $ODT_EXT_END   [LOCAL VARIABLE INSERT] *****//  
#endif  
  
    TRY  
    {  
  
#ifdef ODT_DEFAULT  
    return OK;  
#endif  
  
    }
```

```

#ifdef ODT_EXTENDED
//***** $ODT_EXT_START [EVENT PROCESSING INSERT] *****//
//***** $ODT_EXT_END   [EVENT PROCESSING INSERT] *****//
#endif

    }
    BEGHANDLERS
    CATCHALL {

        return (NOT_OK);

#ifdef ODT_EXTENDED
//***** $ODT_EXT_START [ EXCEPTION HANDLING CATCHALL INSERT] *****//
//***** $ODT_EXT_END   [ EXCEPTION HANDLING CATCHALL INSERT] *****//
#endif

    }
    ENDHANDLERS
}

```

10.10.7 Example Generated Code in .hh File

The `chai_user.hh` file contains class definitions required for implementing behaviors. The object framework uses the methods defined in this file to perform CMIS operations on the managed object instance. The following secretaries are defined in this file:

- `chai_AttrSecty`
- `chai_ActionSecty`
- `chai_InstanceSecty`

10.10.7.1 Generated Object Definitions

```

//-----//
//          OBJECT ATTRIBUTES ENUMERATION          //
//-----//
enum chai_ATTR_INDEX{
    IDX_chaiKettleNumber,
    IDX_chaiBlend,
    IDX_chaiReady,
    NUM_chai_ATTR
};

```



```
//-----//  
//          OBJECT ACTIONS ENUMERATION          //  
//-----//  
enum chai_ACTION_INDEX {  
    IDX_brewChai,  
    NUM_chai_ACTION  
};
```

10.10.7.2 Generated OIDs

```
//-----  
//          ATTRIBUTE OBJECT IDENTIFIERS (OID)          //  
//-----  
#define OID_chai_chaiKettleNumber  "1.3.6.1.4.1.42.2.2.2.1.96.7.1"  
#define OID_chai_chaiBlend         "1.3.6.1.4.1.42.2.2.2.1.96.7.2"  
#define OID_chai_chaiReady         "1.3.6.1.4.1.42.2.2.2.1.96.7.3"  
  
//-----  
//          ACTION OBJECT IDENTIFIERS (OID)             //  
//-----  
#define OID_chai_brewChai          "1.3.6.1.4.1.42.2.2.2.1.96.9.1"  
  
//-----  
//          NAME BINDING OBJECT IDENTIFIERS (OID)        //  
//-----  
#define OID_chai_chai_system       "1.3.6.1.4.1.42.2.2.2.1.96.6.1"  
#define OID_chai_chai_chai         "1.3.6.1.4.1.42.2.2.2.1.96.6.2"
```

10.10.7.3 Attribute Class Definition

```
class chai_AttrSecty: public AttrSecty  
{  
private:  
protected:  
    // Constructor ODT RESERVED  
    chai_AttrSecty(ObjMethMOI &m, const AttrSecty *sp,  
                   const AttrSectyTmpl &t);  
    // Destructor ODT RESERVED  
    ~chai_AttrSecty();  
public:  
    // Local storage for MOI's Attributes  
    Asn1Value chaiKettleNumber;  
    Asn1Value chaiBlend;  
    Asn1Value chaiReady;
```

```

// User Methods
Result read(const AttrSectyInfo &ai, AsnlValue &av);
Result write(const AttrSectyInfo &ai, const AsnlValue &av);
Result fetch(const AttrSectyInfo &ai, const Callback &cb);
Result store(const AttrSectyInfo &ai, const Callback &cb);
Result destroy_vote();
Result receive_event(AsnlValue &event_type, AsnlValue &event_info);

void action(const ActionSectyInfo &ai, const AsnlValue &input,
            const Callback &cb);

static int AttrSectyInfo2index(const AttrSectyInfo &ai);
Result index2AttrSectyInfo(int index, AttrSectyInfo &ai);
Result index2ActionSectyInfo(int index, ActionSectyInfo &ai);

// ODT RESERVED METHODS
static AttrSecty *new_secty(ObjMethMOI &m, const AttrSecty *sp,
                           const AttrSectyTmpl &t);
AsnlValue &index2lhsvalue(int attrindex);
Result delete_prepare(DeleteType type);
static Result is_valid_index(int index);
static Oid index2Oid(int attrindex);
chai_ActionSecty *get_actionsecty()
{
    chai_MOI *mm = (chai_MOI *)&moi;
    chai_ActionSecty *actionsecty = mm->actionsecty;
    return actionsecty;
}

#ifdef ODT_EXTENDED
//***** $ODT_EXT_START [MEMBER FUNCTION/PROTOTYPE INSERT] *****//
//***** $ODT_EXT_END [MEMBER FUNCTION/PROTOTYPE INSERT] *****//
#endif

};

```

10.10.7.4 Action Class Definition

```
class chai_ActionSecty: public ActionSecty
{
protected:
    chai_ActionSecty(ObjMethMOI &m, const ActionSecty *sp,
                     const ActionSectyTpl &t) ;

public:
    // USER METHODS
    void    action(const ActionSectyInfo &ai, const AsnlValue &input,
                  const Callback &cb);

    // INTRA-OBJECT CONV. METHODS
    Result  read(const AttrSectyInfo &ai, AsnlValue &av);
    Result  write(const AttrSectyInfo &ai, const AsnlValue &av);
    Result  fetch(const AttrSectyInfo &ai, const Callback &cb);
    Result  store(const AttrSectyInfo &ai, const Callback &cb);
    Result  index2AttrSectyInfo(int index, AttrSectyInfo &ai);
    Result  index2ActionSectyInfo(int index, ActionSectyInfo &ai);

    // ODT RESERVED METHODS
    static ActionSecty *new_secty(ObjMethMOI &m, const ActionSecty *sp,
                                  const ActionSectyTpl &t)
    {
        return (ActionSecty *)new chai_ActionSecty(m, sp, t);
    }
    static      Result      is_valid_index(int index);
    static      Oid         index2Oid(int attrindex);
    chai_AttrSecty *get_attrsecty()
    {
        chai_MOI *mm = (chai_MOI *)&moi;
        chai_AttrSecty *attrsecty = mm->attrsecty;
        return attrsecty;
    }
    virtual void check(const ActionSectyInfo &ai,
                      const ActionInfo *t,
                      const AsnlValue &av, const Callback &cb);

#ifdef ODT_EXTENDED
    //***** $ODT_EXT_START [MEMBER FUNCTION/PROTOTYPE INSERT] *****//
    //***** $ODT_EXT_END   [MEMBER FUNCTION/PROTOTYPE INSERT] *****//
#endif
};
```

Writing Management Protocol Adaptors (MPAs)

The Solstice EM MIS (Management Information Server) is responsible for maintaining the MIT (Management Information Tree) and ensuring that all activity within the MIT is transparent to an application. This transparency allows applications to make requests for information in a normalized fashion without regard for object location or communications protocol. The MIS resolves the request and routes it to an appropriate entity that is capable of making the correct protocol request. These entities are called *protocol adaptors*. Adaptors exist to map information into the MIT maintained by the MIS. The Management Protocol Adaptors (MPAs) shipped with Solstice EM exist as separate processes from the MIS. Three such adaptors are shipped with Solstice EM: the SNMP MPA, the RPC MPA, the CMIP MPA, and the JDMK MPA. These adaptors provide the protocol translation into the SNMP, RPC, CMIP, and JDMK domains respectively.

This chapter explains how to write MPAs.

- Section 11.1 “Review of MIS Architecture” on page 11-2
- Section 11.2 “Initializing Management Protocol Adaptors and Protocol Driver Modules” on page 11-4
- Section 11.3 “Routing Messages” on page 11-12
- Section 11.4 “MPA/PDM Request Management” on page 11-19
- Section 11.5 “Timer Management” on page 11-23
- Section 11.6 “File Descriptor Management” on page 11-26
- Section 11.7 “Notifications” on page 11-30
- Section 11.8 “Sample MPA/PDM Source Code” on page 11-33
- Section 11.9 “Developing an Adaptor” on page 11-35

11.1 Review of MIS Architecture

The MIS has a modular architecture. Modules are connected by Service Access Points (SAPs). These SAPs provide an asynchronous bi-directional message pipe interface. The core of MIS is the Message Routing Module (MRM) which routes messages to the appropriate modules. New modules can be added to the MIS by attaching a SAP from the new module to the MRM. Each SAP has a unique address which is registered with the MRM when it is attached.

New adaptors are introduced to the system by creating a new SAP to the MRM. (For MPA, no new SAP needs to be created. All MPAs are routed by the same MPA SAP.)

The figure below illustrates some of the modules within the MIS..

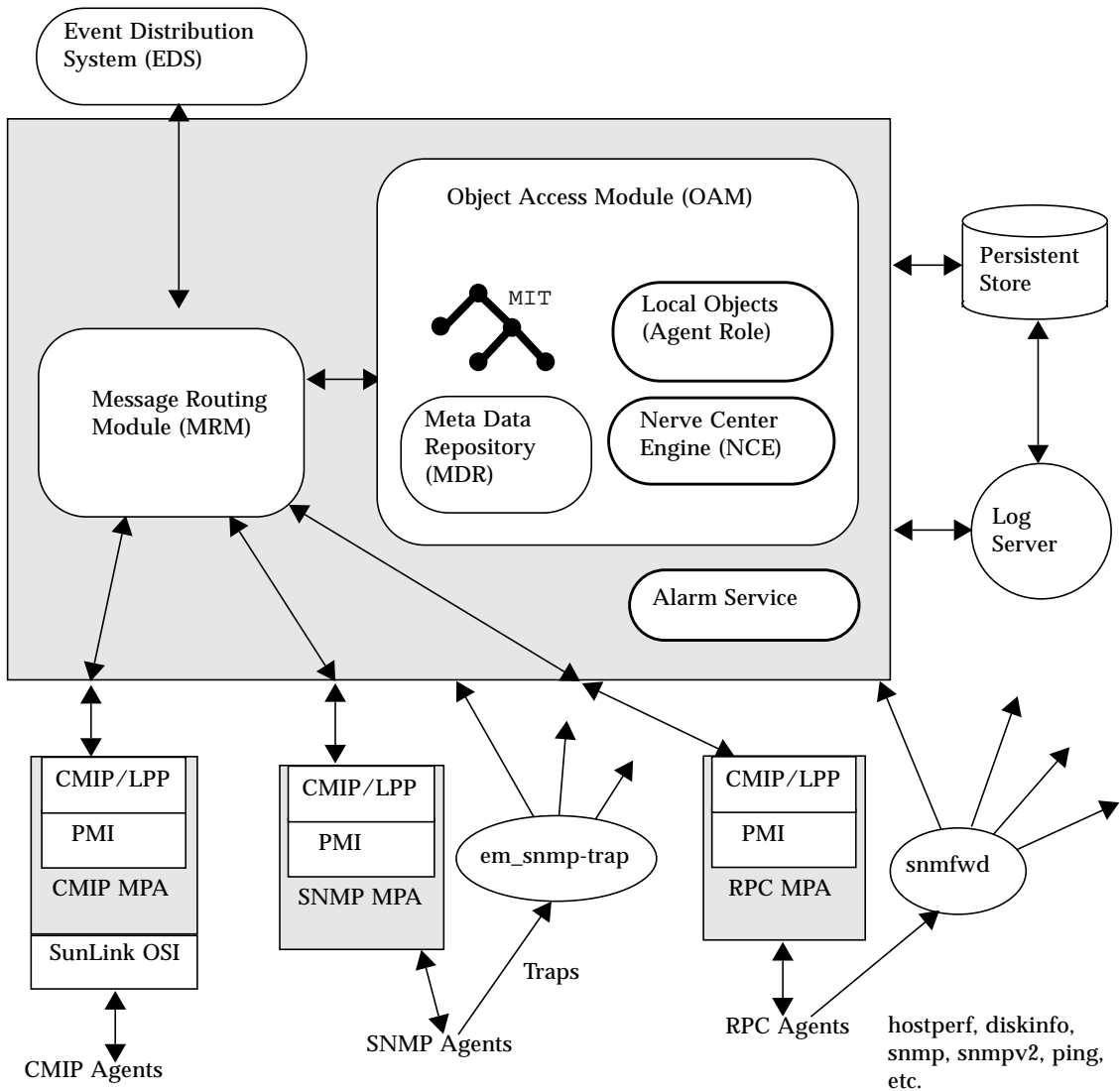


FIGURE 11-1 MIS Architecture

The shaded components in FIGURE 11-1 illustrate how users can add new adaptors to communicate to a proprietary device X. The two choices are clearly indicated, MPA or PDM.

All requests are initially sent to the Message Routing Module (MRM). The MRM uses configuration information maintained in the MIT to determine where to satisfy the request. Local requests are sent to the OAM. Remote requests are directed to the appropriate adaptor that has registered to handle that remote portion of the MIT.

Note – Both the MPA and the PDM use SAP to communicate to the MRM. There is very little difference between a MPA and a PDM once the SAP has been created. They both receive identical messages through the same interface.

11.2 Initializing Management Protocol Adaptors and Protocol Driver Modules

This section describes how Management Protocol Adaptors and Protocol Driver Modules are initialized.

11.2.1 Services Access Points (SAPs)

SAPs provide an asynchronous message passing service. The message set is based on the set of CMIP Protocol Data Units (PDUs). The complete set of messages that can be passed over a SAP is defined in `message.hh`.

All SAP's are C++ class instances derived from the C++ `MessageSAP` class defined in `message.hh`. This class defines the interfaces that allow messages to be sent and received over SAP pairs. An initialized SAP consists of a coupled pair of `MessageSAP` instances.

Note – *SAP* will now be used to refer to an initialized SAP pair.

Once a SAP has been initialized, the following functions may be performed:

```
SendResult      send(MessagePtr mp, MTime block_time = INFINITY);
SendResult      send(MessagePtr mp, const Callback &cb,
                  MTime block_time = INFINITY);
Result receive_request(MessagePtr &mp);

Result receive_response(MessId m_id, MessagePtr &mp);
Result receive_response(ResponseHandle rh, MessagePtr &mp);
void    cancel_callback(MessId m_id);
void    cancel_callback(Callback &cb);
```

This set of functions provides the following message services:

TABLE 11-1 Message Services

Function	Service Provided
Send	Send and expect no Response
Send	Send and schedule to Receive a Response
Receive	Receive a message of a particular id
Receive	Receive the first Message on the incoming queue
Cancel	Cancel callbacks for a particular Message
Cancel	Cancel callbacks for a particular Callback

The interface is defined to be asynchronous. This is achieved using callbacks. Callbacks are a C++ class that consist of a static function pointer and user data. The user data is passed as a parameter to the callback function when the function is invoked.

Example 1

The following example from the `fdn_register` function in `samp_utils.cc` illustrates a CONFIRMED request message.

```
// Need source so responses can come back.
areq->source.aclass = AC_PRIMITIVE;
areq->source.atag = my_sap_no;
Callback recv_cb((CallbackHandler)pdm_receive_resp, sap);
// Send off the request to add.
TRYRES(sap->send(areq, recv_cb, 0));
```

Note – A callback is created with a static function called `pdm_receive_resp` as the callback function pointer. The callback data is the SAP instance pointer. The send function includes the callback `recv_cb` in the send arguments.

When the MIS completes the request, the `pdm_receive_resp` function is called with two parameters. The first parameter is the SAP and the second is a response handle which is used to receive the message response.

Example 2

The following example excerpt is from the `msgio.cc` module and illustrates the `pdm_receive_resp` function.

```
pdm_receive_resp(Ptr cbh, Ptr rh)
{
    Message *mp;
    MessageSAP *p_sap = (MessageSAP *) cbh;

    Vtry {
        TTRYPROC(p_sap->receive_response(rh, mp));
        // At this point mp points to the received response
        Message::delete_message(mp);
    }
}
```

SAPs are initialized by creating SAP pairs. When a SAP pair is created, the `init_kernel_msg_sap` function is used. The `init_kernel_msg_sap` function is a utility function which couples two SAPs. Once the SAPs have been coupled, the local SAP must have two callback functions initialized: the receive request callback and the detached callback.

The receive request callback handler is the handler which is invoked when message requests are sent to the SAP. The detach callback handler is invoked when the remote SAP is deleted.

11.2.2 Initializing a Management Protocol Adaptor

Initializing a Management Protocol Adaptor involves:

- Creating the listen port and request SAP
- Connecting to the MIS and using extract raw SAP
- Locking the application discriminator

11.2.2.1 Creating the Listen Port and Request SAP

The MIS to MPA communication is a form of CMIP over TCP/IP. The MIS creates and manages associations to an MPA on a per request basis. If an association is not used for a period of time the association is released. If an association is already established it will be reused for subsequent requests. Each request has a request identifier which allows for multiple requests to be outstanding at any one time. To facilitate demand based association establishment, the MPA must allocate an IP listen port. An IP listen port is where the MPA listens for association requests from the MIS.

The utility function `init_mpa` creates both a SAP and a listen port.

```
init_mpa(portnum, &p_sap);
```

where `portnumber` is a port number for an IP socket, and `p_sap` is the address of a pointer to a `MessageSAP` data type.

11.2.2.2 Connecting to the MIS and Using *get_raw_sap*

Because the underlying transport mechanism for this SAP might not be active (that is, an underlying association is not established), another SAP must be created to send event reports to the MIS. This other SAP is allocated by creating a `platform` instance which is connected to the MIS. This platform instance gives access to an `ApplMessageSAP`. This SAP is based on another `MessageSAP` implementation. Because the implementation is derived from `MessageSAP`, the `ApplMessageSAP` can be used as a `MessageSAP`. This is possible through C++ inheritance and virtual function mechanisms.

Example

The following code example illustrates how to create a SAP that sends event reports to the MIS and initialize the resultant SAP to receive requests and disconnects.

```
host = getenv("EM_MIS_DEFAULT_HOST");
if (host!=NULL) {
    host = def_host;
}

if (emPlatform.connect(host, "SAMPLE_MPA") == OK) {
    emPlatform.when("DISCONNECTED",
        Callback(pdm_test_handle_detach, 0));
    mis_connected = TRUE ;

} else
    mis_connected = FALSE ;
    ev_sap = (MessageSAP *) emPlatform.get_raw_sap();
    pdm_test_sap->receive_request_cb.handler =

        pdm_receive_request_msgs;
    pdm_test_sap->receive_request_cb.data = (Ptr) pdm_test_sap;
    pdm_test_sap->detach_cb.handler      =
pdm_test_handle_detach;
    pdm_test_sap->detach_cb.data        = (Ptr) pdm_test_sap;
```

Once the platform instance has been created, the `get_raw_sap` method returns the SAP which can be used for sending event reports to the MIS.

11.2.2.3 Locking the Application Discriminator

This step must be completed to allow for proper operation of the MPA. It is most important, as locking the application discriminator has direct impact on system performance. Every platform instance creates an application object instance within the MIS as long as the platform instance is instantiated. Each application object instance contains a discriminator which forwards all platform notifications to the connected application. The MPA typically does not require this feature.

To disable event forwarding, the application instance discriminator must be locked.

Example

The following example excerpt from the `samp_main.cc` module in the `samples` directory illustrates this locking mechanism.

```
DU appinst = emPlatform.get_prop(duAPPLICATION_OBJNAME) ;
Image app_image(appinst);
if( app_image.get_error_type()!=PMI_SUCCESS ||

!app_image.boot() ||
!app_image.set_str("administrativeState","locked") ||
!app_image.store() ) {

    printf("Error starting MPA %s\n",
           app_image.get_error_string());
    exit(1);

}
```

11.2.3 Initializing a Protocol Driver Module

Protocol Driver Modules are shared libraries that are loaded at MIS start-up time. The MIS looks in the `$EM_HOME/config/EM_shared_libs` file for a list of libraries to load. Once it finds an entry in that file, it uses the `dlopen()` system call to load that library. Once the library is loaded, the MIS looks for an instance of the `DynLoader` class that matches the name of the loaded library. Once the `DynLoader` instance has been located, the entry point is invoked with a `D_LOAD` command.

Note – The second parameter of the instance of the declared `DynLoader` must match the name of the shared library. See `dyn_lib.cc` for an example.

Initializing a Protocol Driver Module involves:

- Creating a kernel message SAP pair
- Registering the SAP with the Fully Distinguished Name (FDN) table

11.2.3.1 Creating a Kernel Message SAP

A kernel message SAP pair is created using the `init_kern_msg_sap` function.

```
extern Result init_kernel_msg_sap ( Address , MessageSAP ** );
```

This utility function creates a pair of coupled kernel message SAPs. It returns to the caller a pointer to a local `MessageSAP` and initializes the remote SAP to be attached to the MRM at the SAP Address supplied. This Address must be the same as the Address supplied in the FDN table configuration step. See the section on “Example of Timer Initialization” for more information on Address Format.

Each request (message) contains a destination address field *dest*. The MRM searches in its list of attached SAPs for an address match. When the destination address of a request matches a registered SAP, the request is routed over that SAP.

Note – The destination field component is completed by the lookup code within the MIS.

11.2.3.2 Registering an FDN Table Entry

The utility function `fdn_register` in `samp_utils.cc` illustrates how to add an entry to the FDN table.

```
Result fdn_register(MessageSAP *, int, Asn1Value &);
```

The `fdn_register` function creates a table entry which specifies a mapping between a fully distinguished name (FDN) and a specific address. It takes three parameters to create a table entry:

- A SAP over which to send the `addfdn` ACTION request
- An integer which specifies the SAP TAG (unique identifier for a SAP)
- An encoded distinguished name

Example

The following example from `dyn_libs.cc` illustrates the creation of a table entry:

```
// Check in em_config for TEST SAP number.
// GETENV is a front end MACRO to EM-config file
// Using EM-config forces SAP numbers to be unique
// if not there use default of defined VAL (64)
const char *p_sap_no;

if ( (p_sap_no = GETENV("TEST_PDM_SAP") ) ) {
    pdm_test_addr.atag = atoi(p_sap_no);
} else
    pdm_test_addr.atag = MY_PDM_SAP;

pdm_test_addr.aclass = AC_PRIMITIVE;
pdm_test_sap = (MessageSAP *) 0;

// Create a kernel message SAP, this binds us to the
// MRM at SAP class AC_PRIMITIVE and SAP number MY_PDM_SAP
// Whenever a request's DN matches a DN in the FDN table
// that request is forwarded to the SAP that matches
// the Address portion of the FDN Table Entry. In this case
// our Address is AC_PRIMITIVE, MY_PDM_SAP.

if ( init_kernel_msg_sap ( pdm_test_addr, &pdm_test_sap ) != OK )
    Return(NOT_OK);

// Here is where we set up our request handlers.
pdm_test_sap->receive_request_cb.handler =
    pdm_receive_request_msgs;
pdm_test_sap->receive_request_cb.data = (Ptr) pdm_test_sap;
pdm_test_sap->detach_cb.handler =
    pdm_test_handle_detach;
pdm_test_sap->detach_cb.data = (Ptr) pdm_test_sap;
ev_sap = pdm_test_sap;

// This is where we register the PDM DN with the
// fdn table. We register an entry with
// DN = /systemId="pdm" and an address of
// of AC_PRIMITIVE and SAP number MY_PDM_SAP
```

```
AsnlValue pdm_dn = my_dn(my_name);

// Lets delete first in case we did not unload
// gracefully.
TRYRES(fdn_unregister(pdm_test_sap, MY_PDM_SAP, pdm_dn));

// Now register should work
TRYRES(fdn_register(pdm_test_sap, MY_PDM_SAP, pdm_dn));
```

See Section 11.3.2 “MPA and PDM Addresses” on page 11-14” for information on using `EM_config`. See Section 11.3.1.1 “Address Classes” on page 11-13” for more information on the `AC_PRIMITIVE` class.

11.3 Routing Messages

11.3.1 How Messages are Routed to the Adaptors

The MIS contains a table which is analogous to the NFS mount table that the UNIX kernel maintains. The table is a complete mapping of the remote MIT. The MRM searches this table for complete or partial matches of the Distinguished Name (DN) for every request. If there is a match, the MRM stores the address information found in the FDN table entry in the destination field of the message and forwards the message to the SAP that matches that address.

To route a message to an adaptor the FDN table must previously have been updated with the DN or DNs that the adaptor is responsible for and an address for the adaptor's SAP.

The FDN table is updated by two action requests: `emAddFdnEntry` and `emRemoveFdnEntry`. These action requests are defined in the *em.gdmo* document. The `fdn_register` function uses these actions to update the FDN table. The CMIP Configuration utility (`em_cmipconfig`) uses the same actions to achieve the same purpose.

The FDN table entry has two components: the DN and the Address. The address component is based on the address C++ class defined in `address.hh`. An address has three components:

- *class*, type of Address
- *tag*, particular instance of a particular class
- *value*, data associated with that address class

11.3.1.1 Address Classes

Four address classes are defined:

- AC_DEFAULT

Implies default routing based on type, for example, an event report is always routed to the EMM.

- AC_APP

Is the address class for SAPs that are attached to applications. Each PMI application results in an Application SAP being created. The address class of each application SAP is AC_APP.

- AC_DIR_SERVICE

Is reserved for directory service management.

- AC_PRIMITIVE

Is the address class for PDMs and MPAs

11.3.1.2 AC_PRIMITIVE Address Tags (SAP number)

Each Address class uses the tag value to uniquely identify SAP instances. For the purposes of developing PDMs or MPAs, familiarity with AC_PRIMITIVE tags is critical.

The following is a list of the well known AC_PRIMITIVE tags:

```
#define AT_PRIM_OAM                0
#define AT_PRIM_EMM                1
#define AT_PRIM_CMIP_PRES_ADDR    2
#define AT_PRIM_SNMP_ADDR         3

#define AT_PRIM_AET_ADDR           4    // ASN1: AE-title
#define AT_PRIM_MPA_ADDR           5

#define AT_PRIM_AGENT_DN           6    // ASN1: FDN
#define AT_PRIM_RPC_ADDR           7
#define AT_PRIM_CMIP_CONFIG        8    //
String:{psel,ssel,tset,nsap}
```

Note – The tags listed above are the well known AC_PRIMITIVE tags that are in use by the MIS system. Providers of new PDMs must choose a value outside this range.

11.3.1.3 Address Data (aval)

The data field of an address can be used for any purpose. It is a `DataUnit` which is of variable length. Of particular interest is the `aval` syntax for the `AT_PRIM_CMIP_PRES_ADDR` tagged `AC_PRIMITIVE` class. This contains a complete Presentation Address using the following syntax:

```
length byte, Presentation Selector,  
length byte Session Selector,  
length byte, Transport Selector,  
count byte (number of Network Selectors),  
    length byte, Network Selector.....
```

This is a series of octets, one octet of length followed by value octets. A length value of -1 (255) indicates a null selector.

11.3.2 MPA and PDM Addresses

11.3.2.1 PDM Addresses

A PDM Address is defined to have the following values

- *class*, `AC_PRIMITIVE`
- *tag*, user specified tag, an integer value outside of the range of the well defined set listed above
- *value*, Data Portion is user definable

Example

The following example from `dyn_lib.cc` serves as an illustration:

```
// Check in em_config for TEST SAP number.
// Using EM-config forces SAP numbers to be unique
// if not there use default of defined VAL (64)
const char *p_sap_no;

if ( (p_sap_no = GETENV("TEST_PDM_SAP")) ) {
    pdm_test_addr.atag = atoi(p_sap_no);
} else
    pdm_test_addr.atag = MY_PDM_SAP;

pdm_test_addr.aclass = AC_PRIMITIVE;
pdm_test_sap = (MessageSAP *) 0;
```

Note – A convention exists to ensure that user-supplied PDMs use conflicting PDM SAP numbers. This convention is illustrated above. The convention consists of specifying the PDM number in the `$RUNTIME/conf/EM-config` file which can be extracted using the `GETENV` macro.

11.3.2.2 MPA Addresses

MPAs have one more level of indirection than PDMs. The additional level allows the MPA to be a separate process that can be run anywhere in the TCP/IP domain. The MPA addresses are defined as:

- *class*, `AC_PRIMITIVE`
- *tag*, `AT_PRIM_MPA_ADDR`.

The developer need not worry about configuring an MPA Address programmatically since the Address for an MPA is configured using the `em_cmipconfig` utility. The `em_cmipconfig` utility creates an address where the `aval` portion contains all the relevant configuration information. The MIS extracts this information and presents it to the MPA using the `remote_oi` and `remote` fields of the message request sent to the MPA request routine.

The following fields need to be configured for a custom MPA:

- *Entity Name*
- *Custom MPA*: The port and hostname fields must be completed.
- *Session Selector*

- *Network SAP*: The Network address of the entity containing the real object should be included.
- FDN

Note – If the Presentation address of the remote entity has a null session selector, the session selector must be configured with some default value (for example, “MPA”).

The example MPA supplied used the following configuration:

```
Entity Name : { 1 2 3 4 5 1 }  
Custom MPA port : 5597  
Custom MPA host: "carla"  
  
Session Selector : "Test"  
Network SAP : carla:5597  
FDN : /pdmId=string:"testMPA"
```

Once the `em_cmipconfig` configuration has been completed, the MRM attempts to match entries against what has been configured in the FDN table using `em_cmipconfig`.

Requests for multiple entities can be directed to a custom MPA. The sample source supplied only supports one entity `/pdmId="testMPA"`. A custom MPA can be created to support multiple entities.

The CMIP MPA provided with the MIS is an example of an MPA that is capable of supporting multiple entities. Each CMIP agent in the MIS' management domain is viewed as an entity. For each CMIP agent to be managed, the user must run `em_cmipconfig` specifying the agent parameters, in particular, the agent's presentation address.

11.3.2.3 Message *remote_oi* and *remote* Fields

The values in the `remote_oi` and `remote` fields are one of the specific differences between an MPA and a PDM.

The MPA can make explicit use of these fields. The `remote_oi` contains the DN of the CMIP table entry that was used to route the request to the MPA. The last RDN of this DN can be used to find the AE-Title (Application Entity Table). The `remote`

field contains whatever was configured for that entity's presentation address. This information is what the CMIP MPA uses to establish an association with a remote entity. The `aval` portion of the `remote` field contains a string of the format:

```
"{ Pres, Sess, Tsel, Net }"
```

This is an ASCII representation of the remote entities presentation address as configured using `em_cmipconfig`. See Section 11.3.1.3 “Address Data (`aval`)” on page 11-14 for more information.

Note – A PDM can not make use of the `remote_oi` or the `remote` field.

Multiple FDN table entries can be stored which point to a single MPA or PDM. This storage mechanism can be used as a persistent configuration store for each entity supported by the MPA/PDM. This mechanism, in conjunction with the `remote_oi` and `remote` fields, allows for easy multiplexing to the real object information.

11.3.3 FDN Table Configuration Options

There are multiple ways to configure the MIS to route messages to adaptors. In deciding which configuration to use, it is important to implement a model view that is appropriate for the problem being solved. Common sense can be a good start. For example, it would not be appropriate to add thousands of entries to the FDN table when the MPA could easily implement an efficient proprietary lookup based on the DN.

To help clarify the process, the following options are examined:

- MPA supporting two remote objects
- PDM supporting two remote objects

These examples are used to indicate the various options open to the developer.

Note – These examples are not the only options and are used for illustrative purposes only.

11.3.3.1 MPA Supporting Two Remote Objects

Two possible strategies for adding entries to the FDN table are:

- Two entries can be added to the FDN table using `em_cmipconfig` with two different entity names specified.

- Two entries can be added to the FDN table using `em_cmipconfig` with only one entity name specified.

The second case illustrates how `em_cmipconfig` allows one entity to support multiple objects. It is assumed that the remote entity that supports these objects is identically addressed, that is, the objects live at the same presentation address.

11.3.3.2 PDM Supporting Two Remote Objects

Two possible strategies for adding entries to the FDN table are:

- Two entries can be added to the FDN table with an identical PDM address.
- Two entries can be added to the FDN table with identical class and tag values in the address with different data value in the address value field.

The first case assumes that the DN is decoded in the request and that the request is routed to the appropriate agent entity.

The second case allows the data that is configured in the value (`aval`) portion of the `dest` field of the message to be used for achieving multiplexing to the remote agent entity.

11.3.4 Source and Destination Fields in the Message

All messages contain `src` and `dest` fields. These fields are C++ instances of `Address` and are very important for message routing.

Note – The `src` field must be completed for all CONFIRMED requests.

The MPA/PDM developer must complete the `src` address if responses are expected. The `src` field is the same address used when creating the SAP and when specifying the address component of the FDN table registry entry.

The `dest` field is only important if you wish to explicitly route the message.

Caution – Because explicit routing is used for passing messages between applications, it should be used carefully.

11.4 MPA/PDM Request Management

The MIS has been designed to be a multi-user server and can handle multiple requests transparently and asynchronously. The SAP interface has been designed to make this asynchronous style of programming easier.

In FIGURE 11-2, a scenario is illustrated that is typical of a real world environment.

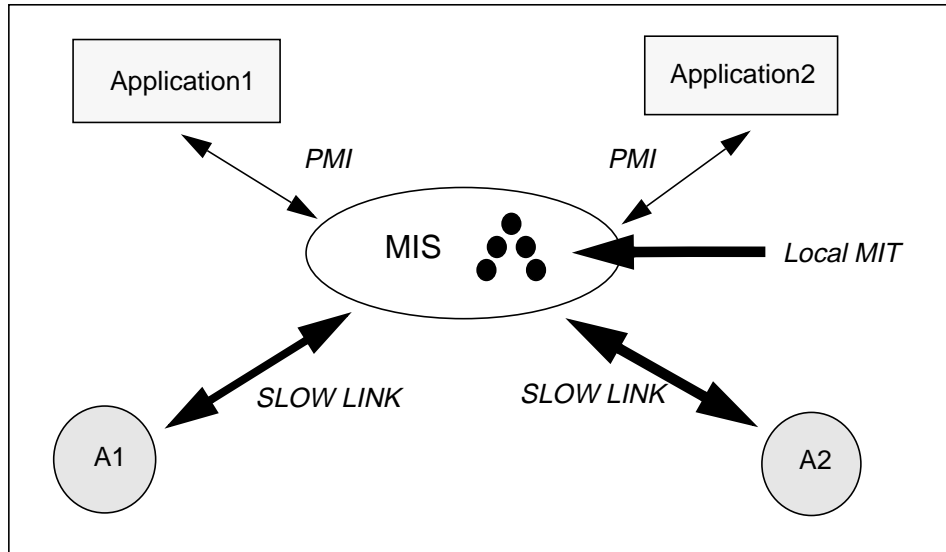


FIGURE 11-2 Potential Real World Configuration

Application1 makes continuous requests to the local MIT. These requests can be satisfied immediately.

Application2 makes requests to A1 and A2. These agents are at the remote end of very slow links. The architecture must support the scheduling of responses to requests that could take a long time to complete. The design and implementation should also handle cases where requests overlap. If appropriate, multiple overlapping requests should be queued and serviced with a single response.

The MPA/PDM module serving A1 and A2 schedules its responses. It achieves this by using the underlying PMI scheduling services.

The transport mechanism used by the MPA/PDM is normally hidden behind a UNIX file descriptor. The MPA/PDM developer needs to use the underlying scheduling services to interface to the file descriptor. This scheduling service makes use of the underlying *poll* (*select*) system call to determine the following:

- Whether data is available at the file descriptor
- Whether data can be written to the file descriptor
- Whether there is an underlying exception on the file descriptor

To satisfy remote requests, the MPA/PDM code typically opens a file descriptor to the agent. As requests are made to the agent, they are written to the file descriptor. The MPA/PDM code then schedules a callback for whenever data becomes available at the file descriptor. When the data is completely transferred, the MPA/PDM sends a response back to the MIS.

At a high level, the steps can be broken down as shown in TABLE 11-2.:

TABLE 11-2 MIS and MPA/PDM Connections

MIS	MPA/PDM
Request for MIS	Create a connection to remote agent. Schedule a callback to indicate connect complete. Send request with unique identifier. Schedule a callback for response. When data callback, check for data complete. If data complete, send response to MIS.
MIS receives Response	

Note – Because all requests have unique identifiers, it is easy to match responses to requests.

11.4.1 Asynchronous Request Code Specifics

When managing asynchronous requests, familiarity with some of the underlying C++ classes that facilitate asynchronous request management is critical. The most important C++ class is the `Callback` class. This is used by all of the asynchronous interfaces. There is also a C++ programming style that must be understood. The proposed C++ model is that a C++ class must be built with a class implementation with the following characteristics:

- Original request information, particularly the request id, the src, and dest fields must be stored.
- A static member function must exist that can be used as a callback handler.
- A variable must exist than can be used to count the number of callbacks expected.
- Class instances must be easily found by the request identifier so that CANCEL GET requests can be serviced in a timely manner.
- Responses must be queued for synchronous requests.

These characteristics determine what can be called a pending request class. This type of class encompasses the functionality required for responding to requests asynchronously.

Example

The following example excerpt from a class called `pdm_pend_req` in `samp_inc.hh` illustrates some key elements associated with asynchronous requests:

```
class pdm_pend_req : public QueueElem {
    ObjReqMess *orig_msg;
    MessId    req_id;
public:

    // hash of all pending requests based on request id.
    Hashdeclare(MessId, pdm_pend_req, hash_MessId,
MessIdcmp,0,0)
    static Hash(MessId, pdm_pend_req) *p_pdm_req_hash;
    I32cbs_pending;
    I32num_in_scope;

    ReqStatus status;
    Result start_req(Callback &req_done);
    // Used in cases where atomic operation requested.
    Queue(RespQElement) resp_q;
}
```

The `pdm_pend_req` class has the following characteristics:

- Maintains a copy of the original request and request id
- Has a callback counter
- Can be queued
- Can be hashed based on a `MessId` type
- Has an asynchronous interface to a `start_req` routine
- Has a callback to be used for invoking operation completions

The function of this class is to remember original request information, to start requests, and to ensure that a final response is sent if required. Requests are completed when there are no more callbacks pending, that is, `cbs_pending == 0`.

The `req_mngt.cc` module supplied in the samples directory illustrates all of these key concepts. The `req_mngt.cc` starts requests, counts the callbacks, and releases resources that are no longer required.

11.4.2 Validating Requests

Each request can be validated. The extent of the validation is dependent on the implementor and the specifics of the implementation. In some cases, it will make no sense to validate requests since much of the intelligence to validate is on the remote agent. The CMIP MPA supplied does little more than store request identifiers and forward the requests to the CMIP agent. The sample source, which is supplied, does some validation for illustrative purposes.

The typical items that can be validated up front are object class and operation types. For illustrations of typical items that can be validated, see the code in `msgio.cc` particularly `pdm_verify_oc` and `pdm_verify_dn`.

11.4.3 Matching Requests to Responses

Every request is identified with an identifier. To send a response to a particular request, set the response message id to the request Id. For completeness the src and dest fields must be completed also.

Example

The following example excerpt from `msgio.cc` `send_resp`. `rmp` is a pointer to a response message. `msg` is a pointer to the original request.

```
rmp->id = msg->id;
rmp->source = msg->dest;
rmp->dest = msg->source;
rmp->qos = msg->qos;
```

11.5 Timer Management

The following timer management services are available to the developer:

- Create and delete a timer
- Start a timer
- Stop a timer

Timers are very useful in the MPA/PDM environment. In particular, they are needed to implement time out strategies. Every request requires a response. In the real world, there are occasions when devices do not respond. The timer facilities are useful to set a timeout callback which can send an `TIMED_OUT` response to the MIS in these types of situations.

Note – MPAs cannot use `TIMED_OUT`, `DEST_UNREACH` or `NO_SUCH_DEST` messages. PDMs may use these message to signify errors. For error conditions like these, the MPA should return a `PROC_FALL` message with a Probable Cause specific error which is defined by the implementor.

Timers can also be used to check on device status. If the device status has changed, the MPA/PDM can emit an attribute change notification. Applications can then be written to wait for these notifications and to operate on an event driven basis. Asynchronous applications (applications that do not `POLL` or that are event driven) have a positive impact on overall system performance.

The MPA/PDM developer should always attempt to use a notification based mechanism to communicate with applications. Timers can be used to schedule these specific notifications. Using GDMO, any type of notification can be designed. Once the GDMO syntax has been loaded into the metadata repository (MDR), the MPA/PDM can emit the notification based on the behavior defined in the GDMO.

11.5.1 Timer Management Interface

The timer interface consist of a set of functions which allow the scheduling of callback handlers based on criteria specified in instances of a timer C++ class. These timer instances specify the following:

- A callback handler
- An interval specifying expiration time before the handler is invoked
- A reload interval value load after the original interval expires

Note – These timers are not suitable for real time solutions.

The class definition for timer is defined in `sched.hh` and is listed below as well as the interface functions to the scheduler.

```
class Timer {
public:
    MTime      time;           // expiration time in milliseconds
    MTime      reload;        // reload time after expiration
    Callback   cb;            // callback to post when expired

    Timer() {
        time = 0;
        reload = 0;
    }
    Timer(MTime t, MTime re, CallbackHandler hand, Ptr d) {
        time = t;
        reload = re;
        cb.handler = hand;
        cb.data = d;
    }

    friend int operator==(const Timer &t1, const Timer &t2) {
        return t1.cb == t2.cb;
    }
};

void    post_timer(const Timer &);
        // Post a timer into the scheduler queue

void    purge_timer(const Timer &);
        // Purge any matching timers from the scheduler queue

void    purge_timer_handler(CallbackHandler handler);
        // Purge any timers with matching handler

void    purge_timer_data(Ptr data);
        // Purge any timers with matching data
```

The four functions detailed above provide the interface to the scheduler for enabling and disabling callbacks based on time. The user can choose a purge interface suitable for the implementation.

11.5.1.1 Example of Timer Initialization

The following example excerpt `start_timer` from `rusagobj.cc` uses the `post_timer` scheduling function to start a timer. The timer that is passed to `post_timer` is an instance of the C++ class `timer`.

```
void start_timer() {  
    // Timer Input is in milli secs  
    // parms are: timer interval, timer reload value  
    // Handler and parm passed to handler  
  
    post_timer(Timer(timerval * 1000, timerval * 1000,  
        (CallbackHandler) pdmrusage_timer, (Ptr) this));  
    timer_posted = TRUE;  
}
```

The timer constructor takes three parameters

- Timer intervals
- Timer reload interval value
- Callback to be executed

All interval values are specified in milliseconds. The timer instance created above is based on a variable, `timerval`, stored in the `rusageobj` instance. This value is in seconds and is converted to milliseconds by multiplying by 1000. The implementation requires that the timer be continuous so a reload value identical to the initial interval is passed for the reload parameter. The callback parameter is passed containing `pdmrusage_timer` as the function to be called and the `rusageobj` instance pointer is passed as the Callback user data.

Note – The implementation of `post_timer` makes a copy of the timer and Callback parameters passed. It is okay to use timers and Callbacks that get destructed.

11.5.2 Stopping a Timer

Timers are stopped or purged using the `purge_timer`, `purge_timer_handler` or `purge_timer_data` utility functions. The `stop_timer` method in `rusageobj.cc`, listed below uses the `purge_timer_handler` function to cancel the `pdmrusage_timer` callback when it is no longer needed, that is, when the *runtime* attribute is set to 0.

```
void stop_timer() {  
    purge_timer_handler((CallbackHandler) pdmrusage_timer);  
    timer_posted = FALSE;  
}
```

`purge_timer_handler` searches the list of active timers for any timer that has a callback handler equal to the value passed. It then purges any timers matching from the timer queue.

Note – It is important to remember if timers have been posted. A common error is that users forget to purge timers. The callback then tries to use data that has been deallocated.

11.6 File Descriptor Management

The normal mechanism of communicating outside of a UNIX process is through a UNIX file descriptor. The services provided by the scheduler to interface to file descriptors are important to understand because most MPA/PDMs use file descriptors to communicate with remote devices.

11.6.1 Asynchronous File I/O

It is important that the MPA/PDM never blocks (makes a system call that does not return immediately). The underlying UNIX File I/O system allows file descriptors to be opened or created in NON-BLOCKING modes. It is most important that any file descriptors be opened in a non-blocking mode.

The most important aspect of non-blocking file I/O is that some operations may not complete. The code must handle properly partial reads/writes. By using the underlying operating *poll* system call (*select* on BSD systems), the scheduler

facilitates non-blocking I/O by providing a set of utility functions to allow for callbacks when particular events happen at a file descriptor. These are based on the standard UNIX set:

- Data can be read
- Data can be written
- Error or exception

Whenever a user expects to read from a file descriptor the user should use `post_fd_read_callback`. If data needs to be scheduled to be written, the user should use `post_fd_write_callback`. To handle error cases there should be a handler for exceptions. This handler can be set using `post_fd_except_callback`.

Callback handlers as discussed above are static functions that are invoked when the event they have been scheduled to service occurs. The file management functions are passed the file descriptor they are managing as parameter two. Parameter one is the data specified when the Callback was created. Oftentimes parameter one is a pointer to a C++ instance that contains the file descriptor and any status associated with it.

The complete set of function prototypes as defined in `sched.hh` are listed below:

```
void    post_fd_read_callback(int fd, const Callback &cb);
void    post_fd_write_callback(int fd, const Callback &cb);
void    post_fd_except_callback(int fd, const Callback &cb);
void    purge_fd_read_callback(int fd);
void    purge_fd_write_callback(int fd);
void    purge_fd_except_callback(int fd);
void    purge_fd_callbacks(int fd);
```

Note – File descriptor callbacks should be purged if the instance associated with the callback is deleted.

11.6.2 Example of a Read Callback Implementation

The example below is taken from the `unixobj.cc` sample source code. There are two items of importance:

- Scheduling the callback
- Callback execution

11.6.2.1 Scheduling the Callback

The following example is taken from the `unxobj.cc` `start_get_req` routine. Once the pipe has been successfully opened, the code creates a callback and schedules the callback function execution when there is data available from the pipe. The read callback handler is scheduled using the `post_fd_read_callback` routine.

The following should be noted:

- The FPTR (`fp`) is converted to a file descriptor using `fileno`. (See the man pages for more information.)
- The callback which is created is initialized to have a function pointer `sh_fetch_input` and a data pointer of this. (This is C++ *this* for instance being operated on).

```
// Set up to get called back when Data is
// available from the pipe.
Callback rd_cb((CallbackHandler)sh_fetch_input, this);
post_fd_read_callback(fileno(fp),rd_cb);
```

The routine `sh_fetch_input` is invoked wherever there is data available at the file descriptor `fileno(fp)`.

11.6.2.2 Callback Execution

The code below is excerpted from `unixobj.cc` `sh_fetch_input`.

The `sh_fetch_input` routine does the following:

- Uses parameter one as a `pdmunixOi` pointer since that is what was defined as the user data parameter when the callback was scheduled.
- Accesses the file descriptor from the instance pointer where it was stored.
- Allocates a temporary dataunit to store the data read.
- Reads the data until there is no more data (for example, EOF). When that happens, it calls the `get_complete` method of the `pdmunixOi` C++ instance.
- If there is more data, it reposts the callback having first saved the data read in `sh_data`. The dataunit catenate is used to add any new data to the end of the old data.

Example

```
static void sh_fetch_input(pdmunixOi *p_obj, int)
{
    int rv;
    DataUnit tmp(RSIZ);

    // Read the data
    if ( !(rv = fread((char *) (const Octet *) tmp, 1,
                      RSIZ, p_obj->fp)) ) {

        // End of pipe, need to encode, save for Persistence
        // and then call the requestor get_complete routine.
        pclose(p_obj->fp);
        p_obj->fp = 0;
        p_obj->get_complete();
        return;

    }

    // Store it into the sh_data Dataunit, just the correct amount
    DataUnit rdata(rv);
    memcpy((void *) (const Octet *) rdata, (const Octet *) tmp,
           rv);

    // Need to get rid of old data, so we do not keep growing
    sh_data
    if ( p_obj->firstdata == TRUE ) {
        p_obj->firstdata = FALSE;
        p_obj->sh_data = DataUnit();
    }
    p_obj->sh_data = catenate(p_obj->sh_data, rdata);

    // Need to get Called Back again so reschedule
    post_fd_read_callback(fileno(p_obj->fp),
                          Callback((CallbackHandler) sh_fetch_input, p_obj));
}
```

This is a very complete example of a read callback handler since it remembers state information (data just read) and reschedules itself.

Note – The callback must be rescheduled if more data is to be read. A common error is the omissions of reposting the callback when the data transfer has not been completed.

11.7 Notifications

Notifications are based on the OSI Management Event Reporting function. All notifications are instances of the CMIP Event Report Request. They contain:

- Object Instance
- Object Class
- Event Time
- Event Type
- Event Information

The ASN.1 syntax for a notification is defined in x711.asn and is listed below:

```
EventReportArgument ::= SEQUENCE {
    managedObjectClass      ObjectClass,
    managedObjectInstance   ObjectInstance,

    eventTime               [5] IMPLICIT GeneralizedTime OPTIONAL,
    eventType               EventTypeId,
    eventInfo               [8] ANY DEFINED BY eventType OPTIONAL
}
```

Note – The event information field is an ASN.1 any defined by value, which is defined by the Event Type field. To generate a notification, the eventInfo must first be constructed according to the syntax defined for that specific notification.

Notifications can be confirmed or unconfirmed. Typically notifications are unconfirmed.

Notifications are generated based on the behavior of the object class being modeled. The standard notification set includes ObjectCreation, ObjectDeletion, and attributeValueChange notifications. New notification types can be defined that are specific to the model being presented to the applications. These new notification types and syntaxes must be loaded into the MDR using the em_gdmo and em_asn1 utilities.

11.7.1 Creating a Notification

Notifications are created by:

- Allocating an event report request message
- Filling in the message fields
- Sending the completed messages to the MIS

These steps are illustrated in the `pdm_issue_notif` routine in `msgio.cc`. Excerpts from that routine are used in the following subsections.

11.7.1.1 Allocating an Event Report Message

Messages are allocated using the `new_message` function. An example of how they are allocated follows:

```
// Allocate Event Report Message
if ((mp = (EventReq *)Message::new_message(EVENT_REPORT_REQ))
    == NULL)

{
    pdm_test_error.print("pdm_issue_notif:
    Not enough memory for "
    "M-Event-Report message\n");
    Return(NOT_OK);
}
```

11.7.1.2 Filling in the Event Report Message Fields

The time and info fields are optional and only need to be completed if the syntax demands that they be completed. All other fields must be filled in. Each of the fields require an encoded `Asn1Value`:

```
mp->mode = UNCONFIRMED;
mp->oc = oc;
mp->oi = oi;

TRYRES(event.encode_oid(TAG_CONT(6), event_type));
mp->event_type = event;
mp->event_info = info;
getGeneralizedTime(mp->event_time);
```

Each field has a specific encoding. This must be adhered to as defined by the definition of an EventReport. See the “Notifications” section for information on syntax. The object class field needs to be encoded TAG CONTEXT 0. The OI needs to be defined according to the specific encoding rules for ObjectInstance. See the x711.asn1 documentation for more information about ObjectInstance.

11.7.1.3 Sending a Notification

All messages are sent to the MIS over an initialized SAP.

```
// Send the Message to the MIS
if (ev_sap->send(mp) != SENT)

{
    // Major System Error
    Message::delete_message(mp);

    pdm_test_error.print("pdm_issue_notif: Could not send "
        "M-Event-Report message\n");
    Return(NOT_OK);
}
```

Unconfirmed event report request messages are sent using the `send` function. Confirmed event report requests also use the `send` function, however, it needs the asynchronous version.

```
SendResult send(MessagePtr mp, const Callback &cb, MTime cd
    block_time)
```

The MIS forwards all event report requests to the Event Distribution System (EDS) where it is discriminated (filtered). The notification is then forwarded to applications that have registered for it.

This routine accepts an already encoded Info parameter. For a complex example of encoding an eventInfo structure, see `pdm_make_attr_chginfo` in `samp_utils.cc`.

11.8 Sample MPA/PDM Source Code

The source code example provided is meant for illustrative and educational purposes. Much of the example code would not be used by a standard MPA/PDM. The sample source provides additional Logical Object Services that would normally be provided in the remote agent. To avoid dependencies on a specific remote agent, simple Logical Object Services are included in the sample source.

Note – All of the sample source code should be studied in detail.

11.8.1 Files and Configuration

The sample source and the GDMO and ASN.1 files are included in the `mpa_samples` directory. There is also a set of netperl scripts which can be used to illustrate the functionality of the MPA/PDM.

Note – The GDMO and ASN.1 files must be loaded before any of the example code is used.

The MPA/PDM sample source consists of the files listed in TABLE 11-3.

TABLE 11-3 MPA Example Files

Filename	Description
Makefile	contains rules
dyn_lib.cc	Portion of Code to create dyn lib entry point and initialization
samp_main.cc	Generates a <code>main</code> for the <code>testmpa</code> program. Attaches to MIS and initializes the MPA event sap.
dynload.hh	Needed by for DynLoader Instance
msgio.cc	Handles message IO from the MIS.
req_mngt.cc	Manages the Requests
samp_inc.hh	Includes and definitions
samp_utils.cc	Utility functions

TABLE 11-3 MPA Example Files (*Continued*)

Filename	Description
lroot.cc	Sample logical Root Object
rusageobj.cc	Sample object which reads /proc and generates rusage info. Can be configured to send Attribute Change notifications.
unixobj.cc	Sample that uses the unix popen command to execute unix commands. Illustrates Async File IO and Object Create Notifications.

The makefile can be used to generate two files, `testmpa` and `testpdm.so`. The `testmpa` file is an executable and `testpdm.so` is a shared library.

11.8.1.1 Sample MPA Configuration: `testmpa`

`Testmpa` is an executable that binds to 5597 on whatever machine it is run on. It manages a logical MIT that begins at `/pdmId="testMPA"`. `em_cmipconfig` must be run to address this MPA. The MPA has a DN of `/pdmId="testMPA"`; it is a CUSTOM MPA that lives at a default port of 5597. You can choose whatever machine name that is needed. Be sure to include a session selector of "test" when using `em_cmipconfig`.

Note – The port number can be overridden by using the environment variable `TEST_MPA_DEFAULT_PORT`. Make sure that this port is entered in `em_cmipconfig` when configuring the MPA.

11.8.1.2 Sample PDM Configuration: `testpdm.so`

`Testpdm.so` is a shared library that can be loaded at platform start-up time. To load `testpdm.so`, edit `$EM_HOME/config/EM_shared_libs`. There are two methods for loading `testpdm.so`:

- Provide a complete pathname in the file.

To do so, include the complete pathname of the shared library in `EM_shared_libs`, e.g. `"/opt/ger/pdmsrc/testpdm.so"`.

- Provide the library name if the library will be placed in `$EM_HOME/lib`.

To do so, include the libname and place the shared library in `$EM_HOME/lib`.

The PDM manages a logical MIT that begins at `/pdmId="testPDM"`, it configures itself to be attached to the MRM at an `AC_PRIMITIVE` SAP type with a SAP number of 64.

Note – PDM SAP numbers can be chosen by the implementor.

To allow for multiple PDMs to exist, this SAP number should be configurable and readable from a file. By convention and default, SAP tag numbers are stored in `/var/opt/SUNWconn/em/conf/EM-config`. The format is `Name : Value`. For example:

```
TEST_PDM_SAP : 64
```

could be added to the file. This entry can be read using `GETENV("TEST_PDM_SAP")`. See the source example in `dyn_lib.cc`.

11.9 Developing an Adaptor

Developing an adaptor involves:

- Defining the management information model
- Designing and implementing the request management interface
- Designing and implementing the protocol code

11.9.1 Defining the Management Information Model

The information model presented to the MIS must be defined in GDMO. If the existing agent already has some GDMO definitions, the task may be easier. In some cases, the existing GDMO definitions may not be adequately abstracted and therefore, will not be suitable. If the existing GDMO model does not provide a sufficient level of abstraction, new definitions should be defined. The GDMO model defined should make every attempt to fully abstract the management problem being solved.

For example, in a case where a device has 5000 ports, the following activities could be defined:

- The device could be defined as one single object with a set of attributes and `ACTIONS` that could be used to access the ports.
- Each port could be defined as an object with multiple attributes.

The model chosen is dependent on the problem being solved. In most cases, the more abstract models can provide for less complex development and better performance for the most common operations. Models can also be optimized to enhance the solution.

The preferred solution in the above scenario would be the single object view. This view would be less complex to implement and require less code. Since there is only one logical object to manage, the overhead in maintaining a logical tree would be minimal and the performance may be better. If the object was defined properly, the applications using the model would also be simpler. They would not have to incur the overhead of managing thousands of objects.

There are no specific rules that can be applied here. Common sense and a clear understanding of the *real* problem and *specific* solution required by the customer are the best guides.

11.9.2 The Request Management Interface

As outlined in the sample source, the interface to the MIS must be completely asynchronous. The interface code must be capable of managing multiple outstanding requests. In addition, it must have minimal impact on the overall system performance. This is critical in the case of the PDM. The request interface can be modeled on the sample source and the `msgio.cc` and `req_mngt.cc` modules can provide the basis for any adaptor.

11.9.3 The Protocol Code

Each device or remote entity must support some type of remote access. Often there are existing libraries that have already implemented a suitable protocol interface to the devices which are to be managed. These interfaces should be reused as much as possible. The most important consideration when reusing existing protocol stack code is that there be no underlying interface element that could block. All code must be asynchronous. When only synchronous interfaces are provided, a layer needs to be built that provides the asynchronous interface.

Controlling Access to Applications and Data

Controlling access to applications and data prohibits unwanted access to critical applications and network components. Without access control, any user of your network management solution can read or modify all your network management and configuration data. The risks of this approach can be devastating when users without the proper authority or expertise modify your network management data or the configuration data of your network management solution. By controlling user access, users are allowed to access only those applications and data they need based on their network management responsibilities and other relevant criteria.

This chapter explains how to control access to applications and data.

- Section 12.1 “Access Control Levels” on page 12-1
- Section 12.2 “Enforcing Predefined Access Control Rules” on page 12-4
- Section 12.3 “Modifying Access Control Information” on page 12-8
- Section 12.4 “Getting Access Control Defaults” on page 12-32
- Section 12.5 “Keeping Event Notifications Private” on page 12-36
- Section 12.6 “Making MPAs Secure” on page 12-40

12.1 Access Control Levels

To enable you make your applications and data secure, Solstice EM provides the following levels of access control:

- Application-level access control
- Application-feature-level access control
- Managed-object-level access control
- Event notification access control
- Management protocol adapter (MPA) access control

12.1.1 Application-Level Access Control

Implement application-level access control if you want your entire application to be inaccessible to some users of the network management solution that your application is a part of. For example, if your application is used for the administration of your network management solution, make the application accessible only to system administrators and inaccessible to network operators.

If you implement application-level access control, make sure that your application gives proper feedback to a user that is denied access to your application.

Before you implement application-level access control, make sure that the application's features are defined and implemented.

Solstice EM enables you to implement application-level access control by either of the methods described in:

- Section 12.2 “Enforcing Predefined Access Control Rules” on page 12-4
- Section 12.3 “Modifying Access Control Information” on page 12-8

12.1.2 Application-Feature-Level Access Control

Implement application-feature-level access control if you want some users to be able to access some, but not all, the features of your application. For example, if your application enables users to monitor, add, modify, and delete network resources, implement application-feature-level access control to allow some users to monitor network resources, but not to add, modify, or delete network resources.

If you implement application-feature-level access control, make sure that your application gives proper feedback if a user is denied access to a feature. Where possible, make sure that your application prevents users from performing operations they do not have permission to perform. In a graphical application, make commands for performing such operations inactive and grayed out.

If you implement application-feature-level access control, make the list of application features available to your system administrator so that the system administrator can grant users access rights to perform various operations.

Before you implement application-feature-level access control, make sure that the application's features are defined and implemented.

Solstice EM enables you to implement application-feature-level access control by either of the methods described in:

- Section 12.2 “Enforcing Predefined Access Control Rules” on page 12-4
- Section 12.3 “Modifying Access Control Information” on page 12-8

12.1.3 Managed-Object-Level Access Control

Implement managed-object-level access control if you want some managed objects to be inaccessible to some users of your network management solution.

Managed-object-level access control denies users access to managed objects regardless of which application they use to try to access the managed objects. If you use application-feature-level access control to deny access to these managed objects, you do not prevent users from accessing the managed objects by using other features of other applications.

If you implement managed-object-level access control, make sure that your application gives proper feedback if a user is denied access to a managed object. In addition, make sure that your application can handle any exceptions or errors thrown if a user is denied access to a managed object.

Solstice EM enables you to implement managed-object-level access control by either of the methods described in:

- Section 12.2 “Enforcing Predefined Access Control Rules” on page 12-4
- Section 12.3 “Modifying Access Control Information” on page 12-8

12.1.4 Event Notification Access Control

Implement event notification access control if you want to ensure that a user’s event logs contain only event notifications emitted by managed objects to which the user has access. By default, all events that the Solstice EM platform receives are written to a user’s event logs, including event notifications from managed objects that the user is normally denied access to.

For information on how to implement event notification access control, refer to Section 12.5 “Keeping Event Notifications Private” on page 12-36.

12.1.5 Management Protocol Adapter (MPA) Access Control

Implement MPA access control if you want some managed objects that are accessed through an MPA to be inaccessible to some users of your application.

If you implement MPA access control, make sure that your application gives proper feedback if a user is denied access to a managed object accessed through an MPA.

For information on how to implement MPA access control, refer to Section 12.6 “Making MPAs Secure” on page 12-40.

12.2 Enforcing Predefined Access Control Rules

Enforce predefined access control rules if you want to control access to applications and data, but do not require your application to modify access control rules that have already been defined.

Solstice EM enables you to enforce predefined access control rules for the following levels of access control:

- Application-level
- Application-feature-level
- Managed-object-level

Enforcing predefined access control rules involves:

- Defining the access control rules
- Enforcing application-level and application-feature-level access control
- Handling denial of access to managed objects

Note – The Solstice EM access control module enforces access control rules defined for managed objects. You do not need to add code to your applications for enforcing managed-object-level access control.

For information on source code examples that show how to enforce predefined access control rules, refer to Section A.5.3 “Password Request Example” on page A-10 and Section A.5.3 “Password Request Example” on page A-10.

12.2.1 Defining Access Control Rules

Access control rules are the basis for denying or granting users access to applications, application features, and managed objects. Access control rules identify:

- User groups to which access controls are to be applied
- Applications, features or managed objects to which access is to be granted or denied
- The policy that determines if access is to be granted or denied

Solstice EM enables you to define access control rules in either of the following ways:

- Interactively
- From the command line

12.2.1.1 Defining Access Control Rules Interactively

Defining access control rules interactively provides immediate verification of the access control rules you define, thereby making it simple to define complex access control rules. To define access control rules interactively, use the Security tool. For information on how to use the Security tool, refer to *Managing Your Network*.

12.2.1.2 Defining Access Control Rules From the Command Line

Defining access control rules from the command line saves time and effort when you need to define large numbers of rules, or when you need to apply the same rules to several different systems. To define access control rules from the command line, use the `em_accesscmd` utility.

The `em_accesscmd` utility defines access control rules in accordance with information supplied in `em_accesscmd` commands. The `em_accesscmd` utility can also read `em_accesscmd` commands from an `em_accesscmd` script, which is a text file.

For information on how to use the `em_accesscmd` utility, refer to *Managing Your Network*.

An `em_accesscmd` script is shown in CODE EXAMPLE 12-1.

CODE EXAMPLE 12-1 `em_accesscmd` Script

```
//  
// Run this script during MIS startup by using the em_accesscmd utility.  
// This script creates access control objects for the "Security" application.  
  
//  
// Create Application  
//  
createApplication "security"          "Security sample graphical application"  
  
//  
// Create "Security sample application" features  
//  
createFeature "security" "View only"      "View/Connect only"  
createFeature "security" "Delete"        "Delete objects"  
  
assignApps "Operator" "security"  
  
// Assign groups to the application features  
  
assignAppFeatures "View Only" "security" "View only"
```

The `em_accesscmd` command script shown in this example defines access control rules as follows:

- The `createApplication` command places an application named `security` under access control. The description of this application is `Security sample graphical application`.
- The `createFeature` command places features of the application named `security` under access control as follows:
 - A feature named `View only` for which the description is `View/Connect only`
 - A feature named `Delete` for which the description is `Delete objects`
- The `assignApps` command grants members of the `Operator` privilege group access to the application named `security`.
- The `assignAppFeatures` command grants members of the `View Only` privilege group access to the feature named `View only` of the application named `security`.

12.2.2 Enforcing Application-Level and Application-Feature-Level Access Control

Enforcing application-level and application-feature-level access control enables your application to verify if the user that started the application has the privileges to use the application and its features. These privileges are granted or denied based on access control rules defined for the application as explained in Section 12.2.1 “Defining Access Control Rules” on page 12-4.

12.2.2.1 Enforcing Application-Level Access Control

Enforcing application-level access control involves:

- Ensuring that the access rules defined for the application deny access to all features of the application
- Verifying that no features are granted to the user who is running the application
- Disabling the application

Verifying that no features are granted to the user who is running the application involves:

- Creating and initializing an instance of the `AuthFeatures` class
- Passing this instance in a call to the `get_authorized_features` function of the `Platform` class
- Verifying that the call to `get_authorized_features` returns an empty list

How to enable or disable an application depends on the application.

12.2.2.2 Enforcing Application-Feature-Level Access Control

Enforcing application-feature-level access control involves:

- Getting a list of features granted to the user who is running the application
- For each feature in the list:
 - Verifying if the user is granted access to the feature
 - Enabling or disabling the feature depending on whether the user is granted access to it

Getting a list of features granted to the user who is running the application involves:

- Creating and initializing an instance of the `AuthFeatures` class
- Passing this instance in a call to the `get_authorized_features` function of the `Platform` class

To verify if a user is granted access to a feature, call the `is_authorized` function of the `AuthFeatures` class. In the call to `is_authorized`, specify the name of the feature.

How to enable or disable a of an application feature depends on the application.

12.2.2.3 Example of Enforcing Application-Level and Application-Feature-Level Access Control

CODE EXAMPLE 12-2 shows code for enforcing application-level and application-feature-level access control.

CODE EXAMPLE 12-2 Controlling Application- and Application-Feature-Level Access

```
...
#include <pmi/hi.hh>           // High Level PMI
...
AuthFeatures feature_list;

    if (!em_mis.get_authorized_features(feature_list)) {
        cout << "Not authorized to run this program!" << endl;
        return 0;
    }
...
    if (!feature_list.is_authorized("Delete"))
        XtSetSensitive(main_window->delete_btn, 0);
    else
        XtSetSensitive(main_window->delete_btn, 1);
...
```

In this example, if the call to `get_authorized_features` returns an empty list, a message warning that the user is denied access to the application is printed and the application is terminated. Otherwise, the application verifies if the user is granted access to the `Delete` feature of the application and does one of the following:

- If the user is denied access to the `Delete` feature, the menu command for choosing this feature is made inactive and grayed out.
- If the user is granted access to the `Delete` feature, the menu command for choosing this feature is made active.

12.2.3 Handling Denial of Access to Managed Objects

If your application is likely to be used in a situation where managed-object-level access control is enforced, ensure that your application takes appropriate action if access to a managed object is denied. To do so, include error checking with every operation on a managed object to verify that access was granted before your application tries to process the result of the operation. If your application is denied access to managed objects it requires to function, make sure that your application provides proper feedback to the user and is terminated gracefully.

If the enforcement action for a managed object is deny without response, your application will receive no indication that it has been denied access to a managed object. In this situation, your application will become blocked indefinitely if it waits for a response. To prevent a denial without response from blocking your application indefinitely, specify a timeout period for all operations on managed objects.

12.3 Modifying Access Control Information

Enable an application to modify access control information if:

- You want the access control applied to applications and data to change while the application is running.
- You want to write a custom security tool.

Solstice EM enables you to modify access control information for the following levels of access control:

- Application-level
- Application-feature-level
- Managed-object-level

To enable an application to modify access control information, you use the access control API. Enabling an application to modify access control information involves:

- Activating access control for the Solstice EM platform
- Adding a user to a privilege group
- Listing all application features under access control
- Adding applications and application features to a privilege group
- Defining a target
- Defining a security rule
- Handling access control errors

For information on source code examples that show how to modify access control information, refer to Section A.5.1 “Access Control API Examples” on page A-9.

12.3.1 Activating Access Control for the Solstice EM Platform

Access control the Solstice EM platform is set active or inactive during installation. If you want to enforce access control in your application, you must ensure that access control is active for the Solstice EM platform.

To determine if access control is active, call the `get_access_control_switch` function of the `ACAccessControlRules` class.

To activate access control, call the `set_access_control_switch` function of the `ACAccessControlRules` class. In the call to `set_access_control_switch`, specify the access control switch status as `emAccessControlOn`.

To deactivate access control, call the `set_access_control_switch` function of the `ACAccessControlRules` class, specifying the access control switch status as `emAccessControlOff`.

Note – Any user who runs a program that activates or deactivates access control must have full access privileges, such as those of a system administrator.

CODE EXAMPLE 12-3 shows code for activating access control for the Solstice EM platform.

CODE EXAMPLE 12-3 Activating Access Control for the Solstice EM Platform

```
...
#include <acapi/accesscontrolrules.hh>           // AC rules
...
ACAccessControlRules access_control_defaults;
```

```

ACAccessControlSwitch access_switch =
    access_control_defaults.get_access_control_switch();

if (access_switch == emAccessControlOff)
{
    access_switch = emAccessControlOn;
    access_control_defaults.set_access_control_switch(access_switch);
}
...

```

In this example, if access control is not active for the Solstice EM platform, the function `set_access_control_switch` is called to activate access control.

12.3.2 Adding a User to a Privilege Group

Access control rules are not set for individual users, but for a group of users called a privilege group. Any user that requires access to secure applications, application features, or managed objects must belong to a privilege group. All users in a privilege group have the same access control privileges. The access control model of Solstice EM allows individual users to belong to more than one group.

Solstice EM provides predefined privilege groups as given in TABLE 12-1.

TABLE 12-1 Predefined Privilege Groups

Privilege Group	Description
Full Access	<p>Provided they are enabled to grant all privileges, users belonging to the Full Access group are permitted to create, modify, and remove access to all Solstice EM tools and managed objects according to any existing rules or settings of the default rule.</p> <p>Without the ability to grant all privileges, users will be able to connect to a remote management information server (MIS), but will not be able to update the security controls.</p> <p>This group is <i>not</i> the same as turning <i>off</i> access control. When access control is turned <i>off</i>, no existing rules limit a user's access to applications and managed objects.</p>
Operators	Users belonging to the Operators group can access specific tools but cannot modify managed data.
View Only	Users belonging to the View Only group can view a restricted set of controlled object data, but they cannot modify the data. Users in this group have access to a restricted set of tools to use for viewing data to which they have access.

Adding a user to a privilege group involves:

- Creating the privilege group
- Creating the user
- Making the user known to the management information server (MIS)
- Adding the user to the privilege group and storing the group

12.3.2.1 Creating a Privilege Group

Before you assign a user to a privilege group, make sure that the privilege group exists.

Creating a privilege group involves:

- Creating and initializing an instance of the `ACGroup` class
- Verifying that the privilege group does *not* exist in the MIS
- Adding the privilege group to the MIS

Creating and Initializing an Instance of the `ACGroup` Class

Create and initialize an instance of the `ACGroup` class to represent the privilege group. In the call to the constructor of the `ACGroup` class, specify the privilege group name. The privilege group name is one of those listed in TABLE 12-1 or the name of a custom privilege group you want to define yourself.

If you create a predefined privilege group, you only need to add users to it.

Verifying That a Privilege Group Does Not Exist in the MIS

Before trying to add a privilege group to the MIS, verify that the group does not already exist in the MIS. To verify that a privilege group does not exist in the MIS, call the `exists` function that the `ACGroup` class inherits from the `ACObject` class.

Adding a Privilege Group to the MIS

Adding a privilege group to the MIS enables the MIS to apply access control to the privilege group. To add a privilege group to the MIS, call the `create` function that the `ACGroup` class inherits from the `ACObject` class.

Note – When you add a privilege group to the MIS, the privilege group is stored only in memory, not persistently. To store the privilege group persistently, store it as described in Section 12.3.2.4 “Adding a User to a Privilege Group and Storing the Group” on page 12-14.

Example of Creating a Privilege Group

Code for creating a privilege group is shown in CODE EXAMPLE 12-4.

CODE EXAMPLE 12-4 Creating a Privilege Group

```
...  
#include <acapi/acgroup.hh>                // ACGroup class  
...  
    RWCString group_name("Operator"); // default group name  
    ACGroup group(group_name);  
    if (!group.exists())  
        group.create();  
...
```

In this example, an instance of the ACGroup class is created and initialized to represent the Operator privilege group. The overloaded NOT (!) operator acts on the call to `exists` to verify that the privilege group does not exist. If the privilege group does not exist, the `create` function is called to add the privilege group to the MIS.

12.3.2.2 Creating a User

Before you assign a user to a privilege group, create the user. To create a user, create and initialize an instance of the ACUser class. The ACUser class represents a user in a privilege group. In the call to the constructor of the ACUser class, specify the login name of the user and, optionally, the full name of the user.

Code for creating a user is shown in CODE EXAMPLE 12-5.

CODE EXAMPLE 12-5 Creating a User

```
...  
#include <acapi/acaccessuserlist.hh>        // ACUser class  
...  
    ACUser user(user_name) ;  
...
```

In this example, a user whose login name is specified by the `user_name` variable is created. The initialization of the `user_name` variable is not shown in this example.

12.3.2.3 Making a User Known to the MIS

Each user in your network management environment must be made known to the MIS to be granted privileges to access applications, application features or managed objects.

Making a user known to the MIS involves:

- Creating an access control list
- Adding a user to the access control list and storing the list

Creating an Access Control List

An access control list contains all users in your network management environment.

To create an access control list, call the `get_access_user_list` function of the `ACInterface` class. The `get_access_user_list` function returns an instance of the `ACAccessUserList` class. The `ACAccessUserList` class represents an access control list.

Code for creating an access control list is shown in CODE EXAMPLE 12-6

CODE EXAMPLE 12-6 Creating an Access Control List

```
...
#include <acapi/acinterface.hh>           // ACInterface class
#include <acapi/acaccessuserlist.hh>      // ACAccessUserList class
...
ACInterface ac_interface;
...
ACAccessUserList user_list = ac_interface.get_access_user_list();
...
```

Adding a User to an Access Control List and Storing the List

After you have created an access control list, add to the list each user in the network management environment. To add a user to the access control list, call the `add_user` function of the `ACAccessUserList` class. In the call to `add_user`, specify the instance of `ACUser` that represents the user you want to add.

Each time you add a user to an access control list, store the list in the MIS. To store the list in the MIS, call the `store` function that the `ACAccessUserList` class inherits from the `ACObject` class.

Code for adding a user to an access control list and storing the list is shown in CODE EXAMPLE 12-7.

CODE EXAMPLE 12-7 Adding a User and Storing an Access Control List

```
...
#include <acapi/acaccessuserlist.hh>      // ACAccessUserList class
...
if (user_list.add_user(user) == TRUE)    // Add the user to the user_list
    user_list.store();                  // Store user_list in the MIS

// Error checking and reporting
if (user_list.get_error_type() == ACC_OK)
{
    Return(OK);
}
if (user_list.get_error_type() == ACC_USER_EXISTS)
{
    cout << user_name << " already exists, will continue ..."
        << endl;
    Return(OK);
}
else
{
    cout << "Error: " << user_list.get_error_string() << endl;
    Return(NOT_OK);
}
...
```

In this example, if the attempt to add the user represented by `user_name` succeeds, the access control list is stored in the MIS. If the user already exists in the access control list, the list is not stored, and a message that states that the user exists is printed. If the attempt to add the user fails for another reason, the list is not stored in the MIS. The reason for the failure is printed.

The initialization of the `user_list` object is shown in CODE EXAMPLE 12-6.

The initialization of the `user_name` variable is not shown in this example.

12.3.2.4 Adding a User to a Privilege Group and Storing the Group

Each user that you want to grant access privileges to must belong to a privilege group. Add a user to a privilege group after you have made the user known to the MIS. To add a user to a privilege group, call the `add_group_member` function on the instance of `ACGroup` that represents the group.

Each time you add a user to a privilege group, store the privilege group persistently in the MIS. To store a privilege group persistently in the MIS, call the `store` function that the `ACGroup` class inherits from the `ACObject` class.

Code for adding a user to a privilege group and storing the group is shown in CODE EXAMPLE 12-8.

CODE EXAMPLE 12-8 Adding a User to a Privilege Group

```
...
#include <acapi/acgroup.hh>                // ACGroup class
...
    if (group.add_group_member(user_name) == TRUE)
        group.store();

// Error checking and reporting
    if (group.get_error_type() == ACC_OK)
    {
        Return(OK);
    }
    if (group.get_error_type() == ACC_USER_EXISTS)
    {
        cout << user_name << " already exists, will continue ..."
              << endl;
        Return(OK);
    }
    else
    {
        cout << "Error: " << group.get_error_string() << endl;
        Return(NOT_OK);
    }
}
```

In this example, if the attempt to add the user represented by `user_name` succeeds, the privilege group is stored in the MIS. If the user is already a member of the privilege group, the privilege group is not stored, and a message that states that the user is a member is printed. If the attempt to add the user fails for another reason, the privilege group is not stored in the MIS. The reason for the failure is printed.

The initialization of the `user_name` variable is not shown in this example.

The initialization of the `group` object is shown in CODE EXAMPLE 12-4.

12.3.3 Listing All Application Features Under Access Control

To determine whether all the features you want to restrict access to are under access control, list all application features under application-feature-level access control.

Listing all application features under application-feature-level access control involves:

- Listing all applications under application-feature-level access control
- For each application listed, listing the features of the application that are under access control

12.3.3.1 Listing All Applications Under Application-Feature-Level Access Control

Listing all applications under application-feature-level access control involves:

- Creating and initializing an instance of the `ACInterface` class
- Calling the `get_application_container` function of the `ACInterface` class to get an instance of the `ACApplicationContainer` class
- Calling the `get_all_applications` function on the instance of `ACApplicationContainer` that the call to `get_application_container` returned

If you want to get a description of each application, call the `get_application_description` function on each application in the list returned by the call to `get_all_applications`.

Code for listing all applications under application-feature-level access control and getting a description of each application is shown in CODE EXAMPLE 12-9.

CODE EXAMPLE 12-9 Listing Applications Under Application-Feature-Level Access Control

```
...
#include <acapi/acapplication.hh>
#include <acapi/acapplicationfeature.hh>
#include <acapi/acinterface.hh>
...
    ACInterface ac_interface;
    ACApplicationContainer app_container =
        ac_interface.get_application_container();

    ACApplicationList app_list =
        app_container.get_all_applications();
```


CODE EXAMPLE 12-9 Listing Applications Under Application-Feature-Level Access Control
(Continued)

```
...
    for (int k=0; k < app_list.entries(); k++)
    {
        ACApplication app(app_list[k].data());
        cout << app_list[k].data();
        cout << app.get_application_description() << endl;
    }
...

```

12.3.3.2 Listing the Features of an Application That Are Under Access Control

After you have listed all applications under application-feature-level access control, get for each application a list of the features that are under access control.

Listing the features of an application that are under access control involves:

- Creating and initializing an instance of the `ACAppFeatureContainer` class
- Calling the `get_all_features` function of the `ACAppFeatureContainer` class to get an instance of the `ACApplicationFeatureList` class

If you want to get a description of each application, call the `get_feature_description` function on each feature in the list returned by the call to `get_all_features`.

Code for listing the features of an application that are under access control and getting a description of each feature is shown in CODE EXAMPLE 12-10.

CODE EXAMPLE 12-10 Listing Application Features Under Access Control

```
...
#include <acapi/acapplication.hh>
#include <acapi/acapplicationfeature.hh>
...
    ACAppFeatureContainer app_features(application_name);
    ACApplicationFeatureList feature_list =
        app_features.get_all_features();
...
    for (int j=0; j < feature_list.entries(); j++)
    {
        ACApplicationFeature feature(application_name,
            feature_list[j].data());
        cout << feature_list[j].data();
    }

```

```
        cout << feature.get_feature_description() << endl;
    }
    ...
}
```

12.3.4 Adding Applications and Application Features to a Privilege Group

Adding applications and application features to a privilege group defines which applications and application features are accessible to the users in the privilege group.

To add an application to a privilege group, call the `add_application` function on the instance of `ACGroup` that represents the group. In the call to `add_application`, specify the name of the application you want to add.

To add an application feature to a privilege group, call the `add_application_feature` function on the instance of `ACGroup` that represents the group. In the call to `add_application_feature`, specify:

- The name of the application that provides the feature you want to add
- The name of the feature you want to add

Note – For information on how to enable your application to enforce application level and application-feature-level access control, refer to Section 12.2.2 “Enforcing Application-Level and Application-Feature-Level Access Control” on page 12-6.

12.3.5 Defining a Target

A target is a collection of management information that is subject to access control. Define a target for each collection of management information that you want to be subject to the same set of security rules for the same user group.

Solstice EM provides predefined targets as given in TABLE 12-2.

TABLE 12-2 Predefined Targets

Name of Target	Description
DenyAccessControlObjectsChange	Pointer to the object /em-name="accessControlContainer"
Connection	Pointer for the instance of the object subsystemid='EM-MIS' which is of type emApplicationinstance
View Only	Pointer to the root of the management information tree (MIT)

Defining a target involves:

- Creating a target
- Defining the list of operations for the target
- Defining the membership of the target
- Storing the target persistently in the MIS

12.3.5.1 Creating a Target

Before you define the membership of a target, make sure that the target exists.

Creating a target involves:

- Creating and initializing an instance of the ACTargets class
- Verifying that the target does *not* exist in the MIS
- Adding the target to the MIS
- Checking for errors

Creating and Initializing an Instance of the ACTargets Class

Create and initialize an instance of the ACTargets class to represent the target. In the call to the constructor of the ACTargets class, specify:

- **The target name.** The target name is one of those listed in TABLE 12-2 or the name of a custom target you want to define yourself. If you create a predefined target, you do not need to define its membership.
- **Optionally, the target type.** The target type specifies the managed object class that defines the target. The target type is one of the following:
 - X741_TARGETS, which specifies the targets managed object class defined in ITU-T X.741 *Objects and Attributes for Access Control*
 - EM_TARGETS, which specifies the emTargets managed object class defined in Solstice EM access control module

The default target type is `X741_TARGETS`.

Verifying That a Target Does Not Exist in the MIS

Before trying to add a target to the MIS, verify that the target does not already exist in the MIS. To verify that a target does not exist in the MIS, call the `exists` function that the `ACTargets` class inherits from the `ACObject` class.

Adding a Target to the MIS

Adding a target to the MIS enables the MIS to apply access control to the target. To add a target to the MIS, call the `create` function that the `ACTargets` class inherits from the `ACObject` class.

Note – When you add a target to the MIS, the target is stored only in memory, not persistently. To store the target persistently, store it as described in Section 12.3.5.4 “Storing the Target Persistently in the MIS” on page 12-25.

Checking for Errors

After you have added a target to the MIS, check for errors. You have to explicitly check for errors because no errors are returned when you verify if the target exists or when you add a target to the MIS. To check for errors, call the `get_error_type` function that the `ACTargets` class inherits from the `ACObject` class. To get the reason for an error, call the `get_error_string` function that the `ACTargets` class inherits from the `ACObject` class.

Example of Creating a Target

Code for creating a target is shown in CODE EXAMPLE 12-11.

CODE EXAMPLE 12-11 Creating a Target

```
...
#include <acapi/actargets.hh> //ACTargets class
...
    ACTargets target(target_name);
    if (!target.exists())
        target.create();
```

CODE EXAMPLE 12-11 Creating a Target (Continued)

```
// Error checking and reporting
    if (target.get_error_type() != ACC_OK)
    {
        cerr << "Error: " << target.get_error_string() << endl;
        exit(3);
    }
    ...
```

In this example, an instance of the `ACTargets` class is created and initialized to represent the target identified by `target_name`. The overloaded `NOT (!)` operator acts on the call to `exists` to verify that the target does not exist. If the target does not exist, the `create` function is called to add the target to the MIS. If an error occurs, a message explaining the reason for the error is printed.

The initialization of the `target_name` variable is not shown in this example.

12.3.5.2 Defining the List of Operations for a Target

The list of operations for a target specifies which management operations on the target are subject to any security rule applied to the target. The operations are permitted or disallowed, depending on the security rule applied to the target. For information on how to define a security rule, see Section 12.3.6 “Defining a Security Rule” on page 12-26.

Defining the list of operations for a target involves:

- Creating the list of operations for a target
- Adding each operation to the list
- Adding the list of operations to the target

To create a list of operations for a target, create an instance of the defined type `ACOperationsList` to represent the list.

To add an operation to a list of operations, call the `insert` function of the defined type `ACOperationsList`. In the call to `insert`, specify the operation. The operation must be one of the operations defined in TABLE 12-3.

TABLE 12-3 Operations for a Target

Operation	Definition
create	Creates a managed object
delete	Deletes a managed object
get	Obtains attribute values from a managed object
replace	Replaces the existing value with a specified value

TABLE 12-3 Operations for a Target *(Continued)*

Operation	Definition
<code>addMember</code>	Adds a value to the current value of a multi-valued attribute
<code>removeMember</code>	Removes a value from the current value of a multi-valued attribute
<code>replacewithDefault</code>	Replaces the existing value with the default value defined in the property list in the <code>ATTRIBUTES</code> construct of the attribute's GDMO specification
<code>action</code>	Performs an action on a managed object
<code>multipleObjectSelection</code>	Selects multiple objects in the MIT to be the subject of a management operation
<code>filter</code>	Applies a filter to a subtree of the MIT

To add a list of operations to a target, call the `set_operations_list` function of the `ACTargets` class.

Code for defining the list of operations for a target is shown in CODE EXAMPLE 12-12.

CODE EXAMPLE 12-12 Defining the List of Operations for a Target

```

...
#include <acapi/actargets.hh> // ACTargets class and
                             // ACOperationsList typedef
...
    ACOperationsList oper_list;

    RWCString operation1("get");
    oper_list.insert(operation1);

    RWCString operation2("replace");
    oper_list.insert(operation2);
...
    target.set_operations_list(oper_list);
...

```

In this example, a list that consists of the operations `get` and `replace` is defined for a target. The `set_operations_list` function is called to add the list to the target. The initialization of the target object is shown in CODE EXAMPLE 12-11.

12.3.5.3 Defining the Membership of a Target

Defining the membership of a target identifies a collection of management information that is subject to access control. In the Solstice EM environment, this collection of management information is an object set that consists of one or more of the following:

- One or more managed objects
- A subtree of the MIT
- All instances of one or more managed object classes

Selecting One or More Managed Objects

If the managed objects you want to be members of a target are distributed throughout the MIT, select them individually.

To select one managed object, call the `add_moi` function of the `ACTargets` class. In the call to `add_moi`, specify the fully distinguished name (FDN), local distinguished name (LDN), or nickname of the managed object.

Selecting more than one managed object involves:

- Creating an instance of the defined type `ACMOIList` to represent the list of managed objects you want to select
- Calling the `insert` function once for each managed object you want to add to the list
- Passing the list in a call to the `set_moi_list` function of the `ACTargets` class

Selecting a Subtree of the MIT

If the managed objects you want to be members of a target are in a subtree of the MIT, select the subtree. To select a subtree, specify:

- The base managed object
- A scope
- A filter

The base managed object is the root object of the subtree you want to select. To specify the base managed object, call the `add_moi` function of the `ACTargets` class. In the call to `add_moi`, specify the FDN, LDN, or nickname of the base managed object.

The scope selects one or more managed objects in the subtree rooted at the base managed object. The scope is defined with reference to the base managed object.

To specify a scope, call the `set_scope` function of the `ACTargets` class. In the call to `set_scope`, create and initialize an instance of the `ACScope` class to represent the scope. In the call to the constructor of the `ACScope` class, specify one of the scope values given in TABLE 12-4.

TABLE 12-4 Scope Values in the Constructor of `ACScope`

Value	Selects
<code>BASE_OBJECT</code>	The base managed object only
<code>ALL_LEVELS</code>	The base managed object and its entire subtree
<code>ALL_LEVELS_EXCEPT_BASE</code>	The entire subtree of the base managed object excluding the base managed object itself
<code>NTH_LEVEL, n</code>	Only level <i>n</i> subordinates of the base managed object, where <i>n</i> is an integer
<code>BASE_TO_NTH_LEVEL, n</code>	The base managed object and all its subordinates to level <i>n</i> , where <i>n</i> is an integer

The filter selects or rejects objects based on the presence and values of specific attributes. The filter is a boolean expression, which may be a single test or a combination of multiple tests. The filter is optional. If you omit it, all managed objects identified by the base managed object and scope are selected.

To specify a filter, call the `set_filter` function of the `ACTargets` class. In the call to `set_filter`, specify an instance of the defined type `ACFilter` to represent the filter. This defined type corresponds to a string. The required format of this string is identical to a filter in a derivation string as defined in Section 6.3.2.3 “Filter” on page 6-7.

Selecting All Instances of One or More Managed Object Classes

To select all instances of a managed object class, call the `add_moc` function of the `ACTargets` class. In the call to `add_moc`, specify the name of the managed object class.

If a managed object class with the same name is defined in more than one of the GDMO documents loaded into the MIS, you must specify in which document the managed object class you are interested in is defined.

To specify the document, prefix the managed object class name with the document name specified in the `MODULE` construct of the managed object’s GDMO specification, for example: `"My Document":reusedMOC`.

Specifying all instances of more than one managed object class involves:

- Creating an instance of the defined type `ACMOCList` to represent the list of managed object classes you want to specify
- Calling the `insert` function once for each managed object class you want to add to the list
- Passing the list in a call to the `set_moc_list` function of the `ACTargets` class

12.3.5.4 Storing the Target Persistently in the MIS

After you have defined the list of operations for a target and the membership of a target, store the target persistently in the MIS. To store a target persistently in the MIS, call the `store` function that the `ACTargets` class inherits from the `ACObject` class.

Code for storing a target persistently in the MIS is shown in CODE EXAMPLE 12-13.

CODE EXAMPLE 12-13 Storing a Target Persistently in the MIS

```
...
#include <acapi/actargets.hh> // ACTargets class and
...
    if (target.add_moc(moc) && target.set_operations_list(oper_list))
    {
        target.store();
    }
...
```

In this example, if the managed object class and list of operations are added to the target, the target is stored persistently in the MIS. The initialization of the `target` object is shown in CODE EXAMPLE 12-11.

12.3.6 Defining a Security Rule

A security rule grants or denies a privilege group access to the management information in a target. Solstice EM provides predefined security rules as given in TABLE 12-5.

TABLE 12-5 Predefined Security Rules

Rule Name	Description
Full Access	Grants the users of “Operators” and “Full Access” groups access to all managed objects
DenyAccesscontrolObjectsChange	Denies the users of the “Operator” group access to change object attributes
View Only	Grants the users of the “View Only” group access to the following objects named: <ul style="list-style-type: none">• “View Only” which allows the users to view data but not change it• “Connection” which allows them to connect to an MIS to view data

Defining a security rule involves:

- Creating the security rule
- Adding a privilege group to the security rule
- Adding a target to the security rule
- Defining the enforcement action of the security rule
- Storing the security rule persistently in the MIS

12.3.6.1 Creating a Security Rule

Before you add a target or a privilege group to a security rule, create the security rule.

Creating a security rule involves:

- Creating and initializing an instance of the `ACRule` class
- Verifying that the security rule does *not* exist in the MIS
- Adding the security rule to the MIS
- Checking for errors

Creating and Initializing an Instance of the `ACRule` Class

Create and initialize an instance of the `ACRule` class to represent the security rule. In the call to the constructor of the `ACRule` class, specify the name of the security rule. The security rule name is one of those listed in TABLE 12-5 or the name of a custom security rule you want to define yourself. If you create a predefined security rule, you do not need to add a target or a privilege group nor define the enforcement action, *unless* you want to extend the rule.

Verifying That a Security Rule Does Not Exist in the MIS

Before trying to add a security rule to the MIS, verify that the security rule does not already exist in the MIS. To verify that a security rule does not exist in the MIS, call the `exists` function that the `ACRule` class inherits from the `ACObject` class.

Adding a Security Rule to the MIS

Adding a security rule to the MIS enables the MIS to enforce the security rule. To add a security rule to the MIS, call the `create` function that the `ACRule` class inherits from the `ACObject` class.

Note – When you add a security rule to the MIS, the security rule is stored only in memory, not persistently. To store the security rule persistently, store it as described in Section 12.3.6.5 “Storing the Security Rule Persistently in the MIS” on page 12-30.

Checking for Errors

After you have added a security rule to the MIS, check for errors. You have to explicitly check for errors because no errors are returned when you verify if a security rule exists or when you add a security rule to the MIS. To check for errors, call the `get_error_type` function that the `ACRule` class inherits from the `ACObject` class. To get the reason for an error, call the `get_error_string` function that the `ACRule` class inherits from the `ACObject` class.

Example of Creating a Security Rule

Code for creating a security rule is shown in CODE EXAMPLE 12-14.

CODE EXAMPLE 12-14 Creating a Security Rule

```
...
#include <acapi/acrule.hh>           // ACRule class
...
    ACRule rule(rule_name);

    if (!rule.exists())
        rule.create();

// Error checking and reporting
    if (rule.get_error_type() != ACC_OK)
    {
        cerr << "Error: " << rule.get_error_string() << endl;
        exit(4);
    }
...
```

In this example, an instance of the `ACRule` class is created and initialized to represent the security rule identified by `rule_name`. The overloaded NOT (!) operator acts on the call to `exists` to verify that the security rule does not exist. If the security rule does not exist, the `create` function is called to add the security rule to the MIS. If an error occurs, a message explaining the reason for the error is printed.

The initialization of the `rule_name` variable is not shown in this example.

12.3.6.2 Adding a Privilege Group to a Security Rule

Adding a privilege group to a security rule specifies the users that the security rule applies to. To add a privilege group to a security rule, call the `add_group` function of the `ACRule` class. In the call to `add_group`, specify the name of the privilege group you want to add.

CODE EXAMPLE 12-15 shows code for adding a privilege group to a security rule.

CODE EXAMPLE 12-15 Adding a Privilege Group to a Security Rule

```
...
#include <acapi/acrule.hh>           // ACRule class
...
```

CODE EXAMPLE 12-15 Adding a Privilege Group to a Security Rule *(Continued)*

```
    ACRule rule(rule_name);  
    ...  
    RWCString group("Operator");  
    rule.add_group(group);  
    ...
```

In this example, a privilege group named `Operator` is added to a security rule. The initialization of the `rule_name` variable is not shown in this example.

12.3.6.3 Adding a Target to a Security Rule

Adding a target to a security rule specifies the collection of management information that the security rule applies to. To add a target to a security rule, call the `add_targets` function of the `ACRule` class. In the call to `add_targets`, specify the name of the target you want to add.

CODE EXAMPLE 12-16 shows code for adding a target to a security rule.

CODE EXAMPLE 12-16 Adding a Target to a Security Rule

```
    ...  
    #include <acapi/acrul.h>           // ACRule class  
    ...  
    ACRule rule(rule_name);  
    ...  
    RWCString target("View Only");  
    rule.add_targets(target);  
    ...
```

In this example, a target named `View Only` is added to a security rule. The initialization of the `rule_name` variable is not shown in this example.

12.3.6.4 Defining the Enforcement Action of a Security Rule

Defining the enforcement action of a security rule specifies the result of an attempt by a member of the security rule's privilege group to perform a management operation on a member of the security rule's target.

To define the enforcement action of a security rule, call the `set_enforcement_action` function of the `ACRule` class. In the call to `set_enforcement_action`, specify the enforcement action. The enforcement action must be one of the enforcement actions defined in TABLE 12-6.

TABLE 12-6 Enforcement Actions

Enforcement Action	Meaning
allow	Allows the requested management operation to be performed
denyWithResponse	Denies the requested management operation and returns the access denied response
denyWithoutResponse	Denies the requested management operation without giving a response
denyWithFalseResponse	Gives a false response and, if the management operation was performed in confirmed mode, returns incorrect management information
abortAssociation	Aborts the association between the manager and the managed entity

CODE EXAMPLE 12-17 shows code for defining the enforcement action of a security rule.

CODE EXAMPLE 12-17 Defining the Enforcement Action of a Security Rule

```
...
#include <acapi/acrule.hh>           // ACRule class
...
    ACRule rule(rule_name);
...
    EnforcementAction enforcement_action = denyWithResponse;
    rule.set_enforcement_action(enforcement_action);
...
```

In this example, the enforcement action of a security rule is set to `denyWithResponse`. The initialization of the `rule_name` variable is not shown in this example.

12.3.6.5 Storing the Security Rule Persistently in the MIS

After you have defined the enforcement action, store the security rule persistently in the MIS. To store a security rule persistently in the MIS, call the `store` function that the `ACRule` class inherits from the `ACObject` class.

CODE EXAMPLE 12-18 shows code for storing a security rule persistently in the MIS.

CODE EXAMPLE 12-18 Storing a Security Rule

```
...
#include <acapi/acrule.hh>           // ACRule class
...
ACRule rule(rule_name);
...
RWCString group("Operator");
RWCString target("View Only");
EnforcementAction enforcement_action = denyWithResponse;

if (rule.add_group(group) && rule.add_targets(target) &&
    rule.set_enforcement_action(enforcement_action))
{
    rule.store();
}
...
```

In this example, if the privilege group and target are added to the security rule, and if the enforcement action is defined, the security rule is stored persistently in the MIS.

12.3.7 Handling Access Control Errors

Users need to know when an attempted access control operation has failed. By providing accurate information on why the operation failed, your applications can ease a user's work by indicating the corrective action required when problems occur. To provide accurate diagnostic information, test for the success of any function call that may, in some circumstances, fail to return the desired result.

To test for the success of a function call, call the `get_error_type` function. When you test for the success of a function call, call the `get_error_string` function to get information on the error and present that information to the user.

Note – The `get_error_type` and `get_error_string` functions of the access control API are similar to the corresponding functions of the `Error` class described in Chapter 4.

If you need to ignore an error and carry on, call the `reset_error` function. Call this function should *before* any valid call is made on an object.

CODE EXAMPLE 12-19 shows code for testing the success of a function call and for providing information on the reason for the failure if the function call fails.

CODE EXAMPLE 12-19 Error Handling Example

```
...
#include <acapi/acgroup.hh>                // ACGroup class
...
    RWCString group_name("Operator"); // default group name
    ACGroup group(group_name);

    if (!group.exists())
        group.create();
...
// Error checking and reporting
if (group.get_error_type() != ACC_OK)
{
    cout << "Error: " << group.get_error_string() << endl;
    exit(3);
}
...
```

12.4 Getting Access Control Defaults

Getting access control defaults enables you to determine if you need to modify any of the predefined security rules provided with the Solstice EM platform. Solstice EM enables you to get the following access control defaults:

- Default enforcement action for all management operations
- Default enforcement action for all events
- List of trusted hosts
- Access control denial granularity
- Access control domain

12.4.1 Getting the Default Enforcement Action for All Management Operations

The enforcement action for a management operation specifies the result of an attempt to perform the management operation.

The management operations to which an enforcement action can be applied are listed in TABLE 12-3. The possible enforcement actions are listed in TABLE 12-6.

To get the default enforcement action for all management operations, call the `get_default_access` function of the `ACAccessControlRules` class. The `get_default_access` function returns a list of value pairs. There is one item in the list for each management operation. Each item in the list consists of the management operation and its enforcement action separated by a space.

CODE EXAMPLE 12-20 shows code for getting the default enforcement action for all management operations.

CODE EXAMPLE 12-20 Getting Default Access Control for All Operations

```
...
#include <acapi/accesscontrolrules.hh           // AC rules
...
    ACAccessControlRules access_control_defaults;
...
    ACDefaultAccess access =
        access_control_defaults.get_default_access();
...
```

12.4.2 Getting the Default Enforcement Action for All Events

The enforcement action for an event specifies the result of an attempt to view the information contained in an event. By default, Solstice EM applies the same enforcement action to all events, regardless of event type.

To get the default enforcement action for all events, call the `get_default_event_access` function of the `ACAccessControlRules` class. The `get_default_event_access` function returns one of the enforcement actions listed in TABLE 12-6.

CODE EXAMPLE 12-21 shows code for getting the default enforcement action for all events.

CODE EXAMPLE 12-21 Getting the Default Enforcement Action for All Events

```
...
#include <acapi/accesscontrolrules.hh           // AC rules
...
    ACAccessControlRules access_control_defaults;
...
    ACDefaultEventAccess event_access =
        access_control_defaults.get_default_event_access();
...
```

12.4.3 Getting a List of Trusted Hosts

A trusted host provides its root user with full access permission to the MIS of the current host whenever that user is running a Solstice EM application or tool.

To get a list of trusted hosts for a Solstice EM system, call the `get_trusted_host_list` function of the `ACAccessControlRules` class.

CODE EXAMPLE 12-22 shows code for getting a list of trusted hosts.

CODE EXAMPLE 12-22 Getting a List of Trusted Hosts

```
...
#include <acapi/accesscontrolrules.hh           // AC rules
...
    ACAccessControlRules access_control_defaults;
...
    ACTrustedHostList trusted_hosts =
        access_control_defaults.get_trusted_host_list();
...
```

12.4.4 Getting the Access Control Denial Granularity

The access control denial granularity defines the level at which access is denied. The possible values of access control denial granularity are given in TABLE 12-7.

TABLE 12-7 Access Control Denial Granularity Levels

Granularity Value	Meaning
request	Access is denied at the request level. An entire request to access one or more managed objects in the MIS is denied if access to one of the managed objects in the request is denied. The request is allowed only when <i>all</i> managed objects in the request are accessible.
object	Access is denied at the managed object level. Access is denied only to managed objects in the request that are not accessible. Access to the remaining managed objects in the request is allowed.
attribute	Access is denied at the attribute level. A request to access a managed object is denied if access to one or more of its attributes is denied. Access to the managed object is allowed only when all the attributes of the managed object are accessible.

To get the access control denial granularity, call the `get_denial_granularity` function of the `ACAccessControlRules` class. The `get_denial_granularity` function returns one of the granularity values given in TABLE 12-7.

CODE EXAMPLE 12-23 shows code for getting the access control denial granularity.

CODE EXAMPLE 12-23 Getting the Access Control Denial Granularity

```
...
#include <acapi/accesscontrolrules.hh           // AC rules
...
    ACAccessControlRules access_control_defaults;
...
    ACDenialGranularity denial_gran =
        access_control_defaults.get_denial_granularity();
...
```

12.4.5 Getting the Access Control Domain

To get the domain that your access control rules govern, call the `get_domain_identity` function of the `ACAccessControlRules` class.

CODE EXAMPLE 12-24 shows code for getting the access control domain.

CODE EXAMPLE 12-24 Getting the Access Control Domain

```
...
#include <acapi/accesscontrolrules.hh           // AC rules
...
ACAccessControlRules access_control_defaults;
...
    cout << "Access control domain is";
    adjust_indentation("Access control domain is");
    cout << access_control_defaults.get_domain_identity() << endl;
...
```

12.5 Keeping Event Notifications Private

Keeping event notifications private ensures that a user's logs contain only event notifications emitted by managed objects to which the user has access. By default, all event notifications that the Solstice EM platform receives are written to a user's logs, including event notifications from managed objects that the user is normally denied access to.

Keeping event notifications private involves:

- Assigning an owner to a log
- Enabling access control for the log server

12.5.1 Assigning an Owner to a Log

The owner of a log is the user who will read the log. Assigning an owner to a log associates the log with the managed objects to which the owner of the log has access. This association enables the log server to perform access control on event notifications. When the log server receives an event notification, the log server verifies if the owner of the log is granted access to the managed object that emitted the event notification. If the owner is granted access to the managed object, the event is logged. Otherwise, the event is not logged.

Assigning an owner to a log involves:

- Creating an auxiliary object for the log
- Verifying that the auxiliary object does *not* exist in the MIS
- Adding the auxiliary object to the MIS
- (Optional) Changing the owner of the log
- Storing the auxiliary object persistently in the MIS

12.5.1.1 Creating an Auxiliary Object for a Log

In the Solstice EM environment, a log is represented by an instance of the `log` class as defined in ITU-T X.735/ISO 10164-6 *Log Control Function*. This class provides no mechanism for assigning an owner to a log. Consequently, to assign an owner to a log, you must create an auxiliary object for the log. An auxiliary object contains an identifier for a log and an identifier for the owner of the log.

To create an auxiliary object for a log, create and initialize an instance of the `ACDBObject` class. The `ACDBObject` class represents a database object, such as a log, on which access control can be enforced. In the call to the constructor of `ACDBObject`, specify the FDN, LDN or nickname of the log.

12.5.1.2 Verifying That an Auxiliary Object Does Not Exist in the MIS

Before trying to add an auxiliary object to the MIS, verify that the auxiliary object does not already exist in the MIS. To verify that an auxiliary object does not exist in the MIS, call the `exists` function that the `ACDBObject` class inherits from the `ACObject` class.

12.5.1.3 Adding an Auxiliary Object to the MIS

Adding an auxiliary object for a log to the MIS enables the log server to use the auxiliary object for performing access control on event notifications. To add an auxiliary object to the MIS, call the `create` function that the `ACDBObject` class inherits from the `ACObject` class.

Note – When you add an auxiliary object to the MIS, the auxiliary object is stored only in memory, not persistently. To store the auxiliary object persistently, store it as described in “Storing the Auxiliary Object Persistently in the MIS” on page 38.

12.5.1.4 (Optional) Changing the Owner of a Log

When you create an auxiliary object for a log, the log server sets the owner of the log to the user name of the user who created the log. To change the owner of a log, call the `set_auxobject_owner` function of the `ACDbObject` class.

In the call to `set_auxobject_owner`, specify:

- The type of the new owner, which must be `USER`
- The user name of the new owner

Note – Any user who runs a program that changes the owner of a log must have full access privileges, such as those of a system administrator.

12.5.1.5 Storing the Auxiliary Object Persistently in the MIS

After you have added an auxiliary object to the MIS, store the auxiliary object persistently in the MIS. To store an auxiliary object persistently in the MIS, call the `store_auxobject` function of the `ACDbObject` class.

12.5.1.6 Example of Assigning an Owner to a Log

CODE EXAMPLE 12-25 shows code for assigning an owner to a log.

CODE EXAMPLE 12-25 Assigning an Owner to a Log

```
...
#include <pmi/sched.hh> // Scheduler for applications with no GUI
#include <acapi/acdbobject.hh> // ACDbObject class
...

ACDbObject *acdbobj_ptr = new ACDbObject(logname, FALSE);

if (!acdbobj_ptr->exists()) {
{
    if (!acdbobj_ptr->create()) {
        cout<<"The auxiliary object has not been created."<<endl;
        exit(1);
    }
    else{
        cout<<"The auxiliary object has not been created."<<endl;
        exit(1);
    }
}
```

CODE EXAMPLE 12-25 Assigning an Owner to a Log (*Continued*)

```
}
//Send and receive the messages.
dispatch_recursive(FALSE);
cout<<"Resetting the auxiliary object."<<endl;

if(!acdbobj_ptr->set_auxobject_owner(USER, owner_id)) {
    cout<<"Failed to reset the auxiliary object."<<endl;
    exit(3);
}
if(!acdbobj_ptr->store_auxobject()) {
    cout<<"Failed to reset the auxiliary object."<<endl;
    exit(3);
}
```

In this example, an auxiliary object for the log identified by `log_name` is created. The overloaded NOT (!) operator acts on the call to `exists` to verify that the auxiliary object does not exist. If the auxiliary object does not exist, the `create` function is called to add the auxiliary object to the MIS. The `set_auxobject_owner` is then called to change the owner of the log to the user identified by `owner_id`. After the owner of the log has been changed, the `store_auxobject` function is called to store the auxiliary object persistently in the MIS.

The initialization of the `log_name` and `owner_id` variables is not shown in this example.

12.5.2 Enabling Access Control for the Log Server

By default, the log server does *not* perform access control on event notifications. To enable the log server to perform access control on event notifications, set the `EM_LOG_MPA_EVENT_ACCESS` environment variable to `TRUE`. To disable access control on event notifications, set the `EM_LOG_MPA_EVENT_ACCESS` environment variable to `FALSE`. To set environment variables for Solstice EM, edit the `/opt/SUNWconn/em/build/acct/EM-config` configuration file.

12.6 Making MPAs Secure

Making MPAs secure protects objects that are accessed through a management protocol adapter (MPA) from unauthorized access. An MPA performs protocol translation required for communication between the Solstice EM MIS and an external entity, such as an agent. Making an MPA secure enables the MPA to enforce access control on the objects it manages.

To make an MPA secure, you use the access control engine API. Making an MPA secure involves:

- Subscribing to access control events
- Creating and initializing an instance of the `ACE` class
- Handling access control events
- Implementing a class derived from `AuxServerUtils`
- Calling access control decision and enforcement functions

For information on source code examples that show how to make an MPA secure, refer to Section A.5.2 “Access Control Engine API Examples” on page A-9.

12.6.1 Subscribing to Access Control Events

To be able to enforce current access control policies on the objects it manages, a secure MPA must be updated with changes to access control information that is stored in the MIS. Changes to access control information in the MIS are communicated by access control events.

A secure MPA needs to receive access control events to be updated with changes to access control information. A secure MPA receives access control events only if it has subscribed to them.

Subscribing to access control events involves:

- Selecting the managed object that represents the MPA’s connection to the MIS
- Setting the `emSpecialEvents` attribute of the managed object to `accessControlEvents`

For information on how to select a managed object, refer to Section 5.3 “Selecting a Managed Object” on page 5-8. To obtain the FDN of the managed object that represents an MPA’s connection to the MIS, call the `get_prop` function of the `Platform` class. In the call to `get_prop`, specify the `APPLICATION_OBJNAME` property.

For information on how to set an attribute of a managed object, refer to Section 5.7 “Setting Attribute Values of an Object” on page 5-19. You have to set the `emSpecialEvents` attribute to the text string `accessControlEvent`. Therefore, call the `set_str` function of the `Image` class to set this attribute in the `Image` instance.

Code for subscribing to access control events is shown in CODE EXAMPLE 12-26.

CODE EXAMPLE 12-26 Subscribing to Access Control Events

```
....
#include <pmi/hi.hh>                                // High Level PMI
...
Platform emPlatform(duEM);
...
extern Debug pdm_test_error;
...
/*
 * Set the special events on the testmpa's application instance
 */
DU appl_obj = emPlatform.get_prop(duAPPLICATION_OBJNAME);
Image appl_inst(appl_obj);
if (!appl_inst.set_str("emSpecialEvents",
                      "{accessControlEvent}"))
{
    pdm_test_error.print(
        "Error in setting special events on the testmpa's
        application instance\n");
    pdm_test_error.print("Reason: %s\n",
                        emPlatform.get_error_string());
    exit (-1);
}
// Store the changes
if (!appl_inst.store())
{
    pdm_test_error.print(
        "Error in setting special events on the topo server's
        application instance\n");
    pdm_test_error.print("Reason: %s\n",
                        emPlatform.get_error_string());
    exit (-1);
}
```

12.6.2 Creating and Initializing an Instance of the ACE Class

Creating and initializing an instance of the ACE class provides an MPA with the services it requires to enforce access control on the objects it manages. An MPA uses an instance of the ACE class to call access control decision and enforcement functions.

An instance of the ACE class loads access control information that is stored in the MIS into an MPA. The loading of this information is done by internal callbacks. Consequently, the loading of this information is complete only when `dispatch_recursive` is called. If functions of any classes in the access control engine (ACE) API are called before `dispatch_recursive` is called, the default behavior is provided.

Create and initialize an instance of the ACE class in the `main` function of the MPA after the MPA has established a connection to the MIS. In the constructor of the ACE class, specify:

- The domain for which access control is to be provided
- A reference to the class that implements the `AuxServerUtils` class (see Section 12.6.4 “Implementing a Class Derived From `AuxServerUtils`” on page 12-44)

To specify the access control domain, create and initialize an instance of the `ACEDomain` class and pass this instance to the constructor of the ACE class. In the constructor of `ACEDomain`, specify:

- The name of the security domain
- The message service access point (SAP) of the security domain

Code for creating and initializing an instance of the ACE class is shown in CODE EXAMPLE 12-27.

CODE EXAMPLE 12-27 Creating and Initializing an Instance of the ACE Class

```
...
#include <ace/ace.hh>           // ACE API
#include "mpapdm_ace.hh"       // AuxServerUtils implementation
...
ACEDomain ace_domain ("AccessControl", ev_sap);
mpapdm_aux_server = new MpapdmAuxServer();
ace_ptr = new ACE(ace_domain, *mpapdm_aux_server);
...
```

In this example, an instance of the `ACEDomain` class is initialized with the domain `AccessControl` and the message SAP identified by `ev_sap`. This instance and a reference to the `MpapdmAuxServer` class are passed to the constructor of the `ACE` class. The `MpapdmAuxServer` class is an implementation of the `AuxServerUtils` class.

The initialization of the `ev_sap` variable is not shown in this example.

12.6.3 Processing Information in Access Control Events

Changes to access control information in the MIS are communicated to an MPA by access control events. An MPA needs to be able to process information in these events to ensure that its access control information is current.

To enable an MPA to process information in access control events, register a callback for these events. The callback is called when the MPA receives an access control event.

To register a callback for access control events, call the `when` function of the `Platform` class. In the call to `when`, you must:

- Specify the event type
- Initialize an instance of the `Callback` class to represent the callback function you want to register

The event type must be `RAW_EVENT`.

To initialize an instance of the `Callback` class, call its constructor in the call to the `when` function. In the call to the constructor of the `Callback` class, specify:

- The name of the callback function. The callback must be one of the following functions of the `ACE` class:
 - `hi_process_ace_event` to process high-level events
 - `lo_process_ace_event` to process low-level events
- A pointer to the data to be passed as an argument to the callback function. You can specify a null pointer if you do not want to pass any data to the callback function.

Code for registering a callback for access control events is shown in CODE EXAMPLE 12-28.

CODE EXAMPLE 12-28 Registering a Callback for Access Control Events

```
...  
#include <pmi/hi.hh>                                //High Level PMI  
...  
emPlatform.when("RAW_EVENT", Callback(ACE::hi_process_ace_event, ace_ptr));  
...
```

In this example, the callback function `hi_process_ace_event` of the `ACE` class is registered so that it is called when the MPA receives an event of type `RAW_EVENT`. The pointer `ace_ptr` specifies data that is passed to the function.

12.6.4 Implementing a Class Derived From `AuxServerUtils`

The `AuxServerUtils` class is an abstract that contains all of the functions that must be implemented for MPA access control. Before creating a secure MPA, you must implement a class derived from the abstract class `AuxServerUtils`.

An example implementation of a class derived from the `AuxServerUtils` class is shown in the source code example file

`/opt/SUNWconn/em/src/mpa_pdm/src/mpapdm_ace.cc`.

12.6.5 Calling Access Control Decision and Enforcement Functions

Access control decision and enforcement functions check the access control rules. Include calls to these functions in a secure MPA whenever a managed object is accessed through the MPA.

Optimizing Performance

The high-level Portable Management Interface (PMI) provides many features that simplify the coding of an application. However, if you need fast response from an application, or if an application is controlling and monitoring a large number of managed objects, you need to tune the application to obtain optimum performance.

This chapter explains how to optimize the performance of your applications.

- Section 13.1 “General Guidelines for Optimizing Performance” on page 13-1
- Section 13.2 “Selectively Activating Image Instances” on page 13-2
- Section 13.3 “Filtering Events” on page 13-3
- Section 13.4 “Writing Your Own Classes to Represent Managed Objects” on page 13-3
- Section 13.5 “Using the Low-Level PMI” on page 13-3

13.1 General Guidelines for Optimizing Performance

The high-level PMI provides many features that simplify the coding of an application. But using these features reduces the performance and efficiency of the application. Tuning at appropriate places improves the performance and efficiency of an application by:

- **Saving memory.** The default behavior of many high-level PMI classes is to cache all information on managed objects in an application. Caching only the information that is relevant to your application saves memory.
- **Reducing network traffic.** The default behavior of many high-level PMI classes is to transfer all information on managed objects between the management information server (MIS) and an application. Transferring only the information that is relevant to your application reduces the network traffic between the MIS and an application

To simplify tuning an application, code each time-critical section as a separate function call. To tune an application coded in this way, all you need do is replace a single function, rather than rewrite the entire application to improve its performance.

13.2 Selectively Activating Image Instances

Selectively activating an Image instance loads information only on attributes that are relevant to your application. By default, information on all attributes of a managed object is loaded when an instance of Image is activated. Loading information on all attributes increases network traffic and the amount of memory the Image instance uses.

To selectively activate an Image instance, call the `boot` function of the Image class, specifying only the attributes your application needs.

For more information on the `boot` function, refer to Section 5.4 “Updating an Image Instance” on page 5-14.

For more information on the Image class, refer to Chapter 5.

Code for selectively activating an Image instance is shown in CODE EXAMPLE 13-1.

CODE EXAMPLE 13-1 Selectively Activating an Image Instance

```
...
#include <pmi/hi.hh>                // High Level PMI
...
    Image channel_image(channel_name.data());
    if (channel_image.get_error_type() != PMI_SUCCESS) {
        cout << channel_image.get_error_string() << endl;
        return FALSE;
    }
...
    Array(DU) attrs;
    attrs = Array(DU)(1);
    attrs[0] = strdup("program");
    if (!channel_image.boot(attrs)) {
        cout << channel_image.get_error_string() << endl;
        return FALSE;
    }
}
```

In this example, only information on the `program` attribute is loaded when the Image instance is activated.

13.3 Filtering Events

Filtering events makes sure that the MIS forwards to your application only the events that are relevant to the application. Filter events to reduce the amount of network traffic between your application and the MIS it is connected to. By default, the MIS forwards all events it receives to your application.

For information on how to filter events, refer to Section 7.4 “Filtering Events” on page 7-16.

13.4 Writing Your Own Classes to Represent Managed Objects

Using an instance of `Image` to store all the data in a managed object simplifies the coding of your applications, but it does require a lot of memory. If memory is scarce, you can save memory by defining your own C++ class to represent managed objects locally in an application. For more information, refer to Section 5.12 “Representing MIS Instances Locally in an Application” on page 5-35.

13.5 Using the Low-Level PMI

Using the `Album` class to perform management operations on object collections increases network traffic and the amount of memory your application instance uses, particularly if your collections contain a large number of managed objects.

To improve the performance of your application, use the low-level PMI to perform operations on object collections. To keep the coding of your application simple, you can still use the high-level PMI for:

- Enabling your application to access managed objects as described in Chapter 3
- Performing operations on individual managed objects as described in Chapter 5

When you use the low-level PMI, the data you pass in function calls must be in ASN.1 format, which is the format that Solstice EM uses internally. If you are unfamiliar with this format, use the high-level PMI to produce a representation of your data in this format by:

- Writing a prototype application by using the high-level PMI

- Monitoring communications between the MIS and your prototype application by using the `em_debug` tool

For information on how to use `em_debug`, refer to Section 15.2 “Monitoring Communications With the MIS” on page 15-7.

When you have the information you need, modify your prototype to use the low-level PMI, copying the format of ASN.1 data types in the messages displayed by `em_debug`.

CODE EXAMPLE 13-2 shows code for getting information from an object collection. This example is equivalent to using the `Album` class to define the membership of an object collection by derivation.

CODE EXAMPLE 13-2 Getting Information From an Object Collection

```
...
#include <pmi/sched.hh>           // Scheduler for applications without a GUI
...
GetReq *msg = (GetReq*)Message::new_message(GET_REQ);
    asap = (ApplMessageSAP *) em_mis.get_raw_sap();

    msg->id = asap->new_id();
    cout << "Deriving from localroot" << endl;
    // This is the actual_class. For get, set, and delete operations the class
    // of the object is not checked if this is given.
    msg->oc = enc("CMIP-1", "ObjectClass", "globalForm:{2 9 3 4 3 42}");

    DU remainder;
    msg->oi = fdn2oi(""); // localroot systemId branch

    if(!msg->oi)
    {
        cout << "fdn2oi failed" << endl;
        exit(1);
    }

    // Scope of all levels selected
    msg->scope = MessScope(NTH_LEVEL, 1);

    char *filter = "item: equality: {objectClass, satellite}";
    msg->filter = enc("CMIP-1", "CMISFilter", filter);

    if (!msg->filter) {
        cout << "Filter encoding failed!" << endl;
    }

    // Set the attribute list to indicate that no attributes are of interest.
```


CODE EXAMPLE 13-2 Getting Information From an Object Collection (Continued)

```
// Only the object instances are received - none of their attributes.
// This is equivalent to an Album object with NO AUTOIMAGE.

// If all attributes are of interest comment the following line.
// This is equivalent to an Album object with AUTOIMAGE.

// If some, but not all, attributes are wanted refer to boot.cc example.
msg->attr_id_list.start_construct(TAG_SET);

Callback mycb(data_return,0);

// For the reply sent to this message, mycb will be called
if(!asap->send(msg, mycb))
{
    cout << "send failed" << endl;;
    exit(2);
}

last_response = 0;
// last_response will be set after all the linked
// replies are received
do
{
    dispatch_recursive(TRUE);
}
while(!last_response);

...
```

In this example, a CMIS Get request is sent to get the FDN of each instance of the satellite managed object class contained by the root object. An instance of the Callback class is initialized to handle the response received to this request. To ensure that the callback is called, `dispatch_recursive` is called in a loop while the application is still waiting for the last response.

CODE EXAMPLE 13-3 shows the definition of the callback handler for processing responses to the request sent in CODE EXAMPLE 13-2.

CODE EXAMPLE 13-3 Callback for Handling Responses to a Get Request

```
...
void data_return(Ptr userdata, Ptr calldata)
{
    Message *msg;
    asap->receive_response(ResponseHandle(calldata), msg);
}
```

CODE EXAMPLE 13-3 Callback for Handling Responses to a Get Request *(Continued)*

```
if(!msg) {
    printf("receive_response failed \n");
    exit(4);
}

GetRes* grmsg ;
switch (msg->type()) {
    case GET_RES:
        grmsg = (GetRes*)msg;
        if (grmsg->oc) {
            // Note: This is stored in Q? A1- Must be stored somewhere,
            // A2- oc2name() etc calls dispatch_re which calls data_return()
            // each call is > 20 frame stack, for 10 it will > 200 etc....
            ocs.enq(new Asn1ValueElem(grmsg->oc));
        }
        if (grmsg->oi) {
            ois.enq(new Asn1ValueElem(grmsg->oi));
        }
        break;
    default:
        printf("Incorrect message type %d\n",msg->type());
        fflush(stdout);
        exit(5);
}
if(!grmsg->linked) {
    last_response = 1;
    printf("Last Respnsen\n");
}
}
...
```

Guidelines for Compiling and Linking Applications

Applications you develop by using the Solstice EM C++ APIs require specific flags to be set at compilation time. You also need to link your applications with the Solstice EM C++ libraries.

This chapter states the compiler version requirements for applications you develop by using the Solstice EM C++ APIs. It lists, for each class in the Solstice EM C++ APIs, the header files you need to include in your application code and the libraries you need to link your applications with. This chapter also explains the flags you must set when you compile applications developed by using the Solstice EM C++ APIs.

- Section 14.1 “Compiler Version Requirements” on page 14-1
- Section 14.2 “Header Files and Libraries” on page 14-1
- Section 14.3 “Options for Locating Header Files and Libraries” on page 14-7
- Section 14.4 “Compilation Flags” on page 14-8

14.1 Compiler Version Requirements

To compile applications you develop by using the Solstice EM C++ APIs, you must use version 5.3 of the Sun™ Workshop C++ compiler.

14.2 Header Files and Libraries

TABLE 14-1 and TABLE 14-2 list for each Solstice EM scheduler and API class:

- The header file you need to include in your application code
- The flags you need to add to the `LDLIBS` entry in your makefile

The header files of Solstice EM are contained in subdirectories of the `/opt/SUNWconn/em/include` directory. The C++ libraries of Solstice EM are provided as shared libraries. They are contained in the `/opt/SUNWconn/em/lib` directory.

TABLE 14-1 Header Files and Libraries for the Solstice EM Schedulers

Scheduler	Header File	LDLIB Flag
sched (Applications with no GUI)	<code>pmi/hi.hh</code>	<code>-lsched</code>
xtsched (GUI applications)	<code>pmi/xtsched.hh</code>	<code>-lxtsched</code>

TABLE 14-2 Header Files and Libraries for the Solstice EM API Classes

Class	Header File	LDLIB Flag
<code>ACAccessControlRules</code>	<code>acapi/accesscontrolrules.hh</code>	<code>-lacapi</code>
<code>ACAccessUserList</code>	<code>acapi/acaccessuserlist.hh</code>	<code>-lacapi</code>
<code>ACAppFeatureContainer</code>	<code>acapi/acapplicationfeature.hh</code>	<code>-lacapi</code>
<code>ACApplication</code>	<code>acapi/acapplication.hh</code>	<code>-lacapi</code>
<code>ACApplicationContainer</code>	<code>acapi/acapplication.hh</code>	<code>-lacapi</code>
<code>ACApplicationFeature</code>	<code>acapi/acapplicationfeature.hh</code>	<code>-lacapi</code>
<code>ACCallback</code>	<code>acapi/accallback.hh</code>	<code>-lacapi</code>
<code>ACContainer</code>	<code>acapi/accontainer.hh</code>	<code>-lacapi</code>
<code>ACDBObject</code>	<code>acapi/acdbobject.hh</code>	<code>-lacapi</code>
<code>ACDBObjectContainer</code>	<code>acapi/acdbobject.hh</code>	<code>-lacapi</code>
<code>ACE</code>	<code>ace/ace.hh</code>	<code>-lace</code>
<code>ACEContext</code>	<code>ace/ace_context.hh</code>	<code>-lace</code>
<code>ACEDecision</code>	<code>ace/ace_decision.hh</code>	<code>-lace</code>
<code>ACEDomain</code>	<code>ace/ace_domain.hh</code>	<code>-lace</code>
<code>ACEMNotificationEmitter</code>	<code>acapi/notificationemitter.hh</code>	<code>-lacapi</code>
<code>ACEMTargets</code>	<code>acapi/acemtargets.hh</code>	<code>-lacapi</code>
<code>ACEReqData</code>	<code>ace/ace_req_data.hh</code>	<code>-lace</code>
<code>ACGroup</code>	<code>acapi/acgroup.hh</code>	<code>-lacapi</code>
<code>ACGroupContainer</code>	<code>acapi/acgroup.hh</code>	<code>-lacapi</code>
<code>ACInterface</code>	<code>acapi/acinterface.hh</code>	<code>-lacapi</code>

TABLE 14-2 Header Files and Libraries for the Solstice EM API Classes *(Continued)*

Class	Header File	LDLIB Flag
ACObject	acapi/acobject.hh	-lacapi
ACRule	acapi/acrule.hh	-lacapi
ACRuleContainer	acapi/acrule.hh	-lacapi
ACScope	acapi/actargets.hh	-lacapi
ACTargets	acapi/actargets.hh	-lacapi
ACTargetsContainer	acapi/actargets.hh	-lacapi
ACUser	acapi/acaccessuserlist.hh	-lacapi
AccessDenied	pmi/message.hh	-lpmi
ActionReq	pmi/message.hh	-lpmi
ActionRes	pmi/message.hh	-lpmi
Address	pmi/address.hh	-lpmi
Album	pmi/hi.hh	-lpmi
AlbumImage	pmi/hi.hh	-lpmi
AppInstComm	app_comm.hh	-lappapi
AppInstObj	app_comm.hh	-lappapi
AppRequest	app_comm.hh	-lappapi
AppTarget	app_comm.hh	-lappapi
Asn1ParsedValue	pmi/asn1_type.hh	-lpmi
Asn1Tag	Both of the following: • pmi/basic.hh • pmi/asn1_val.hh	-lpmi
Asn1Type	pmi/asn1_type.hh	-lpmi
Asn1Value	pmi/asn1_val.hh	-lpmi
AssocReleased	pmi/message.hh	-lpmi
AttrModifier	pmi/hi.hh	-lpmi
AuthApps	pmi/auth_apps.hh	-lpmi
AuthFeatures	pmi/auth_features.hh	-lpmi
AuxServerUtils	ace/ace_aux_server_utils.hh	-lacapi
Blockage	pmi/sched.hh	-lsched
Callback	pmi/callback.hh	-lpmi
CancelGetReq	pmi/message.hh	-lpmi

TABLE 14-2 Header Files and Libraries for the Solstice EM API Classes *(Continued)*

Class	Header File	LDLIB Flag
CancelGetRes	pmi/message.hh	-lpmi
ClassInstConfl	pmi/message.hh	-lpmi
Coder	pmi/hi.hh	-lpmi
Command	pmi/command.hh	-lpmi
Config	pmi/config.hh	-lpmi
CreateReq	pmi/message.hh	-lpmi
CreateRes	pmi/message.hh	-lpmi
CurrentEvent	pmi/hi.hh	-lpmi
DataUnit	pmi/du.hh	-lpmi
DeleteReq	pmi/message.hh	-lpmi
DeleteRes	pmi/message.hh	-lpmi
Dictionary	pmi/dict.hh	-lpmi
DuplicateOI	pmi/message.hh	-lpmi
DupMessageId	pmi/message.hh	-lpmi
EMAgent	topo_api/topo_api.hh	-ltopo_api
EMCmipAgent	topo_api/topo_api.hh	-ltopo_api
EMCmipAgentDn	topo_api/topo_api.hh	-ltopo_api
EMIntegerSet	topo_api/topo_api.hh	-ltopo_api
EMIntegerSetIterator	topo_api/topo_api.hh	-ltopo_api
EMObject	topo_api/topo_api.hh	-ltopo_api
EMRpcAgent	topo_api/topo_api.hh	-ltopo_api
EMRpcAgentDn	topo_api/topo_api.hh	-ltopo_api
EMSnmipAgent	topo_api/topo_api.hh	-ltopo_api
EMSnmipAgentDn	topo_api/topo_api.hh	-ltopo_api
EMStatus	topo_api/topo_api.hh	-ltopo_api
EMTopoNode	topo_api/topo_api.hh	-ltopo_api
EMTopoNodeDn	topo_api/topo_api.hh	-ltopo_api
EMTopoPlatform	topo_api/topo_api.hh	-ltopo_api
EMTopoType	topo_api/topo_api.hh	-ltopo_api
EMTopoTypeDn	topo_api/topo_api.hh	-ltopo_api

TABLE 14-2 Header Files and Libraries for the Solstice EM API Classes *(Continued)*

Class	Header File	LDLIB Flag
EMdataset	grapher/EMgraph.hh	-lemgraphapi
EMdynamicDataset	grapher/EMgraph.hh	-lemgraphapi
EMgraph	grapher/EMgraph.hh	-lemgraphapi
EMstaticDataset	grapher/EMgraph.hh	-lemgraphapi
Err	grapher/EMgraph.hh	-lemgraphapi
Error	pmi/error.hh	-lpmi
ErrorResUnexp	pmi/message.hh	-lpmi
EventReq	pmi/message.hh	-lpmi
GenInt	pmi/genint.hh	-lpmi
GetListErr	pmi/message.hh	-lpmi
GetReq	pmi/message.hh	-lpmi
GetRes	pmi/message.hh	-lpmi
Hash	pmi/hash.hh	-lpmi
HashImpl	pmi/hash.hh	-lpmi
Hdict	pmi/dict.hh	-lpmi
Hrefdict	pmi/dict.hh	-lpmi
Image	pmi/hi.hh	-lpmi
InvalidActionArg	pmi/message.hh	-lpmi
InvalidAttrVal	pmi/message.hh	-lpmi
InvalidEventArg	pmi/message.hh	-lpmi
InvalidFilter	pmi/message.hh	-lpmi
InvalidOI	pmi/message.hh	-lpmi
InvalidOperation	pmi/message.hh	-lpmi
InvalidOperator	pmi/message.hh	-lpmi
InvalidScope	pmi/message.hh	-lpmi
LinkedResUnexp	pmi/message.hh	-lpmi
Message	pmi/message.hh	-lpmi
MessQOS	pmi/message.hh	-lpmi
MessScope	pmi/message.hh	-lpmi
MessageSAP	pmi/message.hh	-lpmi

TABLE 14-2 Header Files and Libraries for the Solstice EM API Classes *(Continued)*

Class	Header File	LDLIB Flag
MissingAttrVal	pmi/message.hh	-lpmi
MistypedArg	pmi/message.hh	-lpmi
MistypedError	pmi/message.hh	-lpmi
MistypedOp	pmi/message.hh	-lpmi
MistypedRes	pmi/message.hh	-lpmi
Morf	pmi/hi.hh	-lpmi
MorfBuilder	extpmi/exthi.hh	-lextpmi
NCAsyncResIterator	nci/nc_api.hh	-lnci
NCParsedReqHandle	nci/nc_api.hh	-lnci
NCTopoInfoList	nci/nc_apiI1.hh	-lnci
NoSuchAction	pmi/message.hh	-lpmi
NoSuchActionArg	pmi/message.hh	-lpmi
NoSuchAttr	pmi/message.hh	-lpmi
NoSuchEvent	pmi/message.hh	-lpmi
NoSuchEventArg	pmi/message.hh	-lpmi
NoSuchMessageId	pmi/message.hh	-lpmi
NoSuchOC	pmi/message.hh	-lpmi
NoSuchOI	pmi/message.hh	-lpmi
NoSuchRefOI	pmi/message.hh	-lpmi
ObjReqMess	pmi/message.hh	-lpmi
ObjResMess	pmi/message.hh	-lpmi
Oid	pmi/oid.hh	-lpmi
OpCancelled	pmi/message.hh	-lpmi
PasswordTty	pmi/password_tty.hh	-lpmi
Platform	pmi/hi.hh	-lpmi
ProcessFailure	pmi/message.hh	-lpmi
Queue	pmi/queue.hh	-lpmi
ReqMess	pmi/message.hh	-lpmi
ResMess	pmi/message.hh	-lpmi
ResourceLimit	pmi/message.hh	-lpmi

TABLE 14-2 Header Files and Libraries for the Solstice EM API Classes *(Continued)*

Class	Header File	LDLIB Flag
ScopedReqMess	pmi/message.hh	-lpmi
SetListErr	pmi/message.hh	-lpmi
SetReq	pmi/message.hh	-lpmi
SetRes	pmi/message.hh	-lpmi
SyncNotSupp	pmi/message.hh	-lpmi
Syntax	pmi/hi.hh	-lpmi
TimedOut	pmi/message.hh	-lpmi
Timer	pmi/sched.hh	-lpmi
UnexpChildOp	pmi/message.hh	-lpmi
UnexpError	pmi/message.hh	-lpmi
UnexpRes	pmi/message.hh	-lpmi
UnrecError	pmi/message.hh	-lpmi
UnrecLinkedId	pmi/message.hh	-lpmi
UnrecMessageId	pmi/message.hh	-lpmi
UnrecOp	pmi/message.hh	-lpmi
ViewerAPI	viewer_api.hh	-lviewer_api
Waiter	pmi/hi.hh	-lpmi

14.3 Options for Locating Header Files and Libraries

To enable the compiler to locate the required Solstice EM header files and libraries, specify the following entries in your makefile:

```
CFLAGS += -I/opt/SUNWconn/em/include
LDFLAGS += -L/opt/SUNWconn/em/lib -R/opt/SUNWconn/em/lib
```

14.4 Compilation Flags

Set the compilation flags listed in TABLE 14-3 to instruct the compiler to load the proper code for the various operating systems, and to specify whether tracing and debugging are required.

TABLE 14-3 Compilation Flags for Applications Developed With the Solstice EM C++ APIs

Flag	Definition
-DSOLARIS	Specifies the Solaris™ operating environment.
-DSYSV	Specifies the Berkeley BSD System V operating system.
-DNO_TRACE	Disables the tracing functions in the portable management interface (PMI). Set this option for the production version of your applications. It is not needed during the development phase of your applications.
-DDEBUG	Enables use of the PMI debug class. Set this option during the development phase of your applications. It is not needed for the production version.

Troubleshooting

Applications will not always behave as expected. Therefore, good testing and debugging practices are an essential part of any serious software engineering project. During the final stages of development, you need to find and correct all errors in your applications. Such errors typically result from logical errors or simple typing mistakes in the code. In addition, you need to make sure that changes to the environment in which your applications run do not cause your deployed applications to fail.

This chapter provides guidelines on how to troubleshoot errors specific to applications developed by using the Solstice EM C++ APIs.

- Section 15.1 “Testing and Debugging Programs” on page 15-1
- Section 15.2 “Monitoring Communications With the MIS” on page 15-7
- Section 15.3 “Avoiding Common Problems” on page 15-19
- Section 15.4 “Example Troubleshooting Scenarios” on page 15-23

15.1 Testing and Debugging Programs

The Solstice EM C++ development environment provides tools to help you test and debug your programs. Use these tools for:

- Verifying Guidelines for the Definition of Managed Objects (GDMO) and Abstract Syntax Notation One (ASN.1) syntax and logic
- Trapping errors in high-level Portable Management Interface (PMI) function calls
- Trapping programming logic errors
- Monitoring protocol translation by a management protocol adapter (MPA)
- Reloading GDMO documents

15.1.1 Verifying GDMO and ASN.1 Syntax and Logic

Verify that the GDMO and ASN.1 syntax and logic of your function calls are correct if your application fails to:

- Perform create, set, get, or delete operations
- Send an action request

To verify the GDMO and ASN.1 syntax and logic of your function calls, use the MIS Objects tool. The MIS Objects tool enables you to perform operations directly on managed objects in the management information tree (MIT). Use the MIS Objects tool to verify GDMO and ASN.1 syntaxes in your function calls by performing the operation your application is attempting. If you can perform the operation by using the MIS Objects tool, make sure that your program uses the same syntax and logic when it performs the same operation.

15.1.2 Trapping Errors in PMI Function Calls

The classes of the high-level Portable Management Interface (PMI) inherit error handling methods from the `Error` class. To detect the failure of a high-level PMI call, call the `get_error_type` function. The error type set by a failed call provides valuable information about what caused the failure.

Use the functions and operators inherited from the `Error` class to print a diagnostic message if a function call fails. Use the `get_error_string` function of the `Error` class to display information about the specific error. Printing other information, such as the name of the function that failed and the values of parameters passed to the function, also provides information useful in trapping errors in PMI function calls.

For more information, refer to Chapter 4.

15.1.3 Trapping Programming Logic Errors

Programming logic errors result when a program does not execute the correct code in response to an input. Programming logic errors frequently arise when runtime conditions are different from those envisaged when the application was designed.

To trap programming logic errors, trace the code that the application executes. Tracing the code that the application executes determines the exact point at which the program logic differs from the expected behavior, for example, an `if` statement is present where an `else` statement is required.

Trace the code that the application executes by using:

- Text output functions, such as `cout` or `printf`, of standard C++ libraries
- A standard debugging tool such as `dbx` or the GUI-based debugger embedded in the Forte™ (formerly Sun WorkShop™) C or Forte C++ software

If you are using the Forte compilers and want to use a standard debugging tool, you must set the `-g` flag when you compile your programs.

15.1.4 Monitoring Protocol Translation by an MPA

An MPA performs protocol translation required for communication between the Solstice EM MIS and an external entity, such as an agent. Solstice EM provides MPAs for the following protocols:

- Simple Network Management Protocol (SNMP)
- Common Management Information Protocol (CMIP)
- Remote procedure call (RPC) protocol

To obtain detailed information about the protocol translation that the MPAs perform, run them in debug mode.

The MPAs of Solstice EM are started by the `/etc/rc2.d/S98ipmpa` script. To run the MPAs in debug mode, modify this script to specify a debug flag, then restart the MPAs. Modify the script to turn off debug mode when you no longer require information about the protocol translation that the MPAs perform.

The possible debug flags are:

- `-debug` – Shows SNMP, RPC, and CMIP translations
- `-debug2` – Shows all translations specified by the `-debug` flag and shows user datagram protocol (UDP) packets

Note – Only the root user usually has write access to the `/etc/rc2.d/S98ipmpa` script. If you do not have write access to this script, ask your system administrator to give you write access to it.

To specify a debug flag, change the occurrence of `$EXE_PATH/$1` in the `else` statement of the `startmpa()` function to read:

```
$EXE_PATH/$1 flag >/tmp/dbgout_$1 2>&1
```

Where *flag* is `-debug` or `-debug2`.

The translation information from each MPA is written to the following files:

- /tmp/dbgout_em_mpa_snmp (for output from the SNMP MPA)
- /tmp/dbgout_em_mpa_rpc (for output from the RPC MPA)

To stop and restart the MPAs after you have edited the script, type the following commands as root:

```
# /etc/rc2.d/S98ipmpa stop
... shutdown messages ...
# /etc/rc2.d/S98ipmpa start
... startup messages ...
```

15.1.5 Reloading GDMO Documents

During development, particularly during the early phases of development, the GDMO documents that describe your object model are likely to change. If a GDMO document that is already loaded into the metadata repository (MDR) changes, you need to reload the GDMO document into the MDR.

Reloading GDMO documents is rarely required for deployed applications. When equipment is added to your network, a GDMO document that describes the equipment is usually supplied with the equipment. All you need to do is load this document into the MDR.

When you load a GDMO document into the MDR, any version that already exists is not overwritten. To make your new version effective, you must remove the existing version before loading the new version.

You can remove a GDMO document by:

- **Rebuilding the Solstice EM database.** Rebuild the Solstice EM database only if you have no data in the Solstice EM database that you want to preserve.
- **Removing a single compiled GDMO document.** Remove a single compiled GDMO document if you want to preserve data in the Solstice EM database.

15.1.5.1 Removing a GDMO Document by Rebuilding the Solstice EM Database

Removing a GDMO document by rebuilding the Solstice EM database guarantees that your GDMO document is removed from the MDR, but the entire Solstice EM database is destroyed and rebuilt in the process. Rebuild the Solstice EM database only if you have no data in the Solstice EM database that you want to preserve.

To Remove a GDMO Document by Rebuilding the Solstice EM Database

1. **Ensure that the uncompiled GDMO document file is *not* present in the /opt/SUNWconn/em/etc/gdmo directory.**

If necessary, delete the file from the /opt/SUNWconn/em/etc/gdmo directory. For example, to remove the `satman.gdmo` document file, type the following commands:

```
prompt% cd /opt/SUNWconn/em/etc/gdmo
prompt% rm satman.gdmo
```

Note – If you want to modify a GDMO document instead of removing it, modify the uncompiled GDMO document file instead of deleting the file.

2. **Type the command for rebuilding the Solstice EM database:**

```
prompt% em_services -reload
```

When the `em_services -reload` command is run:

- The management information server (MIS) is stopped.
- All data in the Solstice EM database is removed.
- All files contained in the /opt/SUNWconn/em/etc/gdmo directory are loaded into the MDR.
- The MIS is restarted.

For more information on maintaining the MIS, refer to the *Management Information Server (MIS) Guide*.

15.1.5.2 Removing a Single Compiled GDMO Document

Removing a single compiled GDMO document preserves data in the remainder of the Solstice EM database, but requires more work than rebuilding the Solstice EM database. Remove a single compiled GDMO document if you have existing data in the Solstice EM database that you want to preserve.

Note – If you are using Solstice EM in an agent role and have set agent role behavior for Solstice EM, you must rebuild the Solstice EM database. You cannot remove a single compiled GDMO document. For information on how to set agent role behavior for Solstice EM, refer to Section 2.6.2 “Setting Agent Role Behavior of Solstice EM” on page 2-37.

To remove a single compiled GDMO document, locate the file that contains the compiled GDMO document on the MIS host and delete the file.

The name and location of the file that contains a compiled GDMO document depend on the options set when the document was loaded in to the MDR:

- If an output directory was specified, the file is located in the directory specified. The name of the file is the GDMO document name as specified in the `MODULE` construct of the GDMO document. Each space in the document name is replaced with an underscore. For example, a GDMO document named Satellite Manager would be compiled into a file named `Satellite_Manager`.
- If no output directory was specified, the file is located in the `/var/opt/SUNWconn/em/data/MDR` directory on the MIS host. The name of the file is assigned automatically by Solstice EM. The file name begins with the characters `MA`.

To locate a compiled GDMO document the name of which was automatically assigned, use the `grep` command to find all files the names of which start with the characters `MA` that contain the document name. For example, to remove the compiled Satellite Manager GDMO document, type the following commands:

```
host1% rlogin mishost
Password: *****
mishost% cd /var/opt/SUNWconn/em/data/MDR
mishost% rm `grep -l Satellite\ Manager MA*`
mishost% em_services -start
```

After you have removed or replaced a compiled GDMO document, restart the MIS by typing `em_services -start`. This command restarts the MIS without rebuilding the Solstice EM database.

15.2 Monitoring Communications With the MIS

Monitoring communications with the management information server (MIS) enables you to verify if requests sent from your application and notifications emitted by managed objects reach the MIS. Monitoring communications with the MIS provides information on the content of:

- Messages the MIS receives
- Management requests your application sends to the MIS
- Responses sent by managed objects to management requests
- Notifications emitted by managed objects

To monitor communications with the MIS, use the `em_debug` utility. The `em_debug` utility connects to the debug port of the MIS and receives the debugging messages you specify when you start `em_debug`.

15.2.1 Starting `em_debug`

When you start `em_debug`, you can select the types of message you want to display, or exclude message types. When you exclude message types, all currently running `em_debug` sessions stop showing the excluded message types.

To start `em_debug` and select message types, type:

```
prompt% em_debug [-host hostname] [-port port] on messageType
```

To start `em_debug` and exclude message types, type:

```
prompt% em_debug [-host hostname] [-port port] off messageType
```

Where:

- *hostname* is the name of the host on which the MIS is running. If *hostname* is not specified, the local host is assumed.
- *port* is the debug port of the MIS. The default is 5556.
- *messageType* is the type of message that you want `em_debug` to display or exclude. Wildcard characters are permitted in *messageType*.

Commonly used `em_debug` message types are given in TABLE 15-1. For a complete list, see Section 15.2.3 “Full List of `em_debug` Message Types” on page 15-15.

TABLE 15-1 Commonly Used `em_debug` Message Types

Message Type Parameter	Messages Printed
<code>actmsg_debug</code>	PMI action request and response messages
<code>oammsg_debug</code>	Object access module (OAM) get, set, create, and delete messages
<code>asn_debug</code>	ASN.1 module processing messages
<code>oamnotif_debug</code>	Event notification information
<code>snmp_debug</code>	SNMP object access
<code>rpc_debug</code>	RPC object access
<code>mdr_debug</code>	MDR access
<code>emm_info</code>	Event monitor information
<code>cmip_debug</code>	Debug messages from <code>acse</code> , <code>rose</code> , <code>lpp</code> and <code>tcp</code>
<code>cmip_info</code>	CMIP information
<code>ace_debug</code>	Access control information
<code>misc_out</code>	Nerve Center debug messages

Specify the wildcard character (*) if you want to specify several message types in the command to start `em_debug`. For example, to specify that `em_debug` displays all messages related to the OAM, type:

```
prompt% em_debug "on oam*"
```

15.2.2 Interpreting `em_debug` Messages

Interpret `em_debug` messages to find the data related to your application and identify the information your application exchanges with the MIS.

The messages displayed by `em_debug` contain ASN.1 data encoded by using rules specified in ITU-T X.209/ISO-8825 *Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1)*. Values that are encoded by using ASN.1 basic encoding rules have the following fields:

- Tag identifies the type of data that is contained in the `Value` field.
- Len indicates the number of octets used for the `Value` field.
- Value contains a data value of the type specified by the `Tag` field that has a length in bytes as specified by the `Length` field.

A tag is a combination of the class and the tag number. For example, a context-specific class that uses the value 7 for a specific meaning is displayed as `C7`. Application class tags are displayed as `A1`, `A2`, `A3`, etc. Private tags are displayed as `P1`, `P2`, `P3`, etc. Some of the Universal class tags are displayed by using text identifiers. For example, an object identifier (OID) tag is displayed as `OID` rather than as `U6`.

15.2.2.1 Classes of Tags

ASN.1 defines the following classes of tags:

- **Universal class.** Universal class tags are defined by the ASN.1 standards: ITU-T X.208/ISO-8824 *Specification of Abstract Syntax Notation One (ASN.1)* and ITU-T X.209/ISO-8825 *Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1)*. Each universal class tag is assigned to a unique data type or is assigned to a data type that is used to construct new data types. Each tag number shown in TABLE 15-2 is a universal class tag.
- **Application class.** Application class tags are assigned to data types by other international standards. An application class tag is unique within a standard. An example of an application class tag is the `IpAddress` data type used for Internet network management applications, which is defined in an Internet request for comments (RFC).
- **Context-specific class.** Context-specific class tags are interpreted within the context in which they are used. They are normally used with constructed data types such as `SET` or `SEQUENCE`. In the case of `SET` or `SEQUENCE`, a context-specific tag only has meaning within the context of a previously defined `SET` or `SEQUENCE`.
- **Private class.** Private class tags are not assigned by international standards. Private class tags can be used by enterprises to define proprietary data types.

15.2.2.2 ASN.1 Data Types and Tag Numbers

A data type is a type of value, such as integer, real, or string. ASN.1 defines a number of data types and assigns a *tag number* to identify each of the data types. ASN.1 data types and tag numbers are shown in TABLE 15-2.

TABLE 15-2 ASN.1 Data Types and Tag Numbers

Tag Number	ASN.1 Data Types
1	BOOLEAN
2	INTEGER
3	BIT STRING
4	OCTET STRING
5	NULL
6	OBJECT IDENTIFIER
7	OBJECT DESCRIPTOR
8	EXTERNAL
9	REAL
10	ENUMERATED
11-15	Reserved for future use
16	SEQUENCE, SEQUENCE-OF
17	SET, SET-OF
18	NumericString
19	PrintableString
20	TeletexString
21	VideoTexString
22	IA5String
23	UTCTime
24	GeneralizedTime
25	GraphicsString
26	VisibleString
27	GeneralString
28	CharacterString
29	Reserved for future use

15.2.2.3 Tips For Reading `em_debug` Output

Note – In addition to the messages transmitted between applications and the MIS, `em_debug` displays messages that the MIS generates internally.

After you understand how to read `em_debug` output, search for output from your application and identify information that has been exchanged with the MIS.

Output from `em_debug` contains all communication with the MIS and some internally generated messages. If other applications are running, you will see messages from them in addition to messages from your application. You will have to sift through the messages to find what is relevant to you. With a little experience, you will learn how to focus on the messages for your application.

Each request is followed by a response. Whether you are debugging the request or the response from the MIS, start by searching for the request from your application. After you find the request, search for the next matching response if you need it.

For example, if you are debugging a failed create request, search through the `em_debug` output until you find a create request. Then check the managed object specified in the request to see if the create request you are looking at is the one your application sent.

After you find your create request, check the corresponding create response.

The example in Section 15.2.2.4 “Example `em_debug` Output” on page 15-11 shows you how to read `em_debug` output.

15.2.2.4 Example `em_debug` Output

CODE EXAMPLE 15-1 shows an example of the output from `em_debug`. In this example, `em_debug` was started by typing

```
prompt% em_debug -c "on oammsg_debug"
```

CODE EXAMPLE 15-1 Sample `em_debug` Output

```
oammsg_debug: *****
oammsg_debug: received by OAM: create request
oammsg_debug: message type = create request
oammsg_debug: id = 87
```

CODE EXAMPLE 15-1 Sample em_debug Output (*Continued*)

```

oammsg_debug:      source =
oammsg_debug:      aclass = DEF, atag = 0
    aval = Du: no data unit allocated
oammsg_debug:      dest =
oammsg_debug:      aclass = DEF, atag = 0
    aval = Du: no data unit allocated
oammsg_debug:      remote =
oammsg_debug:      aclass = DEF, atag = 0
    aval = Du: no data unit allocated
oammsg_debug:      mode = CONFIRMED
oammsg_debug:      app_context = Du: no data unit allocated
oammsg_debug:      oc =
oammsg_debug:      Tag  Len Value
oammsg_debug:      C0    c 1.3.6.1.4.1.42.2.2.2.1.3.2
oammsg_debug:      oi = NULL
oammsg_debug:      superior_oi =
oammsg_debug:      Tag  Len Value
oammsg_debug:      C4    13
oammsg_debug:      Tag  Len Value
oammsg_debug:      SET   11
oammsg_debug:      Tag  Len Value
oammsg_debug:      SEQ    f
oammsg_debug:      Tag  Len Value
oammsg_debug:      OID    5 2.9.3.5.7.11
oammsg_debug:      Tag  Len Value
oammsg_debug:      GRPH   6 "EM-MIS"
oammsg_debug:      access = NULL
oammsg_debug:      reference_oi = NULL
oammsg_debug:      attr_list =
oammsg_debug:      Tag  Len Value
oammsg_debug:      C7    69
oammsg_debug:      Tag  Len Value
oammsg_debug:      SEQ    11
oammsg_debug:      Tag  Len Value
oammsg_debug:      C0    c 1.3.6.1.4.1.42.2.2.2.1.7.1
oammsg_debug:      Tag  Len Value
oammsg_debug:      INT    1 = 22U
oammsg_debug:      Tag  Len Value
oammsg_debug:      SEQ    1d
oammsg_debug:      Tag  Len Value
oammsg_debug:      C0    c 1.3.6.1.4.1.42.2.2.2.1.7.2
oammsg_debug:      Tag  Len Value
oammsg_debug:      GRPH   d "Object Editor"
oammsg_debug:      Tag  Len Value
oammsg_debug:      SEQ    29
oammsg_debug:      Tag  Len Value

```

CODE EXAMPLE 15-1 Sample em_debug Output (Continued)

```
oammsg_debug:      C0      c 1.3.6.1.4.1.42.2.2.2.1.7.6
oammsg_debug:      Tag   Len Value
oammsg_debug:      SEQ    19
oammsg_debug:                  Tag   Len Value
oammsg_debug:                  SET    17
oammsg_debug:                  Tag   Len Value
oammsg_debug:                  SEQ    15
oammsg_debug:                  Tag   Len Value
oammsg_debug:                  OID     c 1.3.6.
1.4.1.42.2.2.2.1.7.7
oammsg_debug:                  Tag   Len Value
oammsg_debug:                  GRPH   5 "kevin"
oammsg_debug:      Tag   Len Value
oammsg_debug:      SEQ    a
oammsg_debug:                  Tag   Len Value
oammsg_debug:      C0      5 0x59 03 02 07 1f
oammsg_debug:      Tag   Len Value
oammsg_debug:      ENUM    1 = 0
oammsg_debug:      *****
```

In the example in CODE EXAMPLE 15-1, the following message indicates that an application has just started and connected to the MIS:

```
received by OAM: create request
```

Several lines later in the em_debug output, the keyword `oc =` is followed by the OID of the managed object class specified in the create request as follows:

```
oammsg_debug:      oc =
oammsg_debug:      Tag   Len Value
oammsg_debug:      C0      c 1.3.6.1.4.1.42.2.2.2.1.3.2
```

This OID is defined in the GDMO documents supplied with Solstice EM. It is the OID of the `emApplicationInstance` managed object class.

On the next line of the em_debug output, the keyword `oi =` is followed by the name of managed object instance specified in the create request as follows:

```
oammsg_debug:      oi = NULL
oammsg_debug:      superior_oi =
oammsg_debug:      Tag   Len Value
```

```

oammsg_debug:      C4      13
oammsg_debug:      Tag Len Value
oammsg_debug:      SET    11
oammsg_debug:      Tag Len Value
oammsg_debug:      SEQ     f
oammsg_debug:      Tag Len Value
oammsg_debug:      OID     5 2.9.3.5.7.11
oammsg_debug:      Tag Len Value
oammsg_debug:      GRPH    6 "EM-MIS"

```

In this example, `oi = NULL`, indicating that the request is from a call to the `create_within` function of the Image class.

The keyword `superior_oi` is followed by the name of the superior object specified in the create request. The superior object is identified by the local distinguished name `subSystemId="EM-MIS"`. The MIS assigns the relative distinguished name of the new object automatically.

The keyword `attr_list` is followed by each attribute that is set in the create request and the value it is set to as follows:

```

oammsg_debug:      attr_list =
oammsg_debug:      Tag Len Value
oammsg_debug:      C7      69
oammsg_debug:      Tag Len Value
oammsg_debug:      SEQ    11
oammsg_debug:      Tag Len Value
oammsg_debug:      C0      c 1.3.6.1.4.1.42.2.2.2.1.7.1
oammsg_debug:      Tag Len Value
oammsg_debug:      INT     1 = 22U
oammsg_debug:      Tag Len Value
oammsg_debug:      SEQ    1d
oammsg_debug:      Tag Len Value
oammsg_debug:      C0      c 1.3.6.1.4.1.42.2.2.2.1.7.2
oammsg_debug:      Tag Len Value
oammsg_debug:      GRPH    d "Object Editor"

```

The OIDs of the attributes are defined in the GDMO documents supplied with Solstice EM. These OIDs are as follows:

- 1.3.6.1.4.1.42.2.2.2.1.7.1 is the OID of the `emApplicationID` attribute.
- 1.3.6.1.4.1.42.2.2.2.1.7.2 is the OID of the `emApplicationType` attribute.

The `emApplicationID` attribute is set to the INTEGER value 22. The `emApplicationType` attribute is set to the GraphicString "Object Editor".

15.2.3 Full List of em_debug Message Types

The full list of em_debug message types is given in TABLE 15-1.

TABLE 15-3 em_debug Message Types

Category	Message Types
Access control	ac_init_debug
	access_debug
	access_error
	ace_debug
	ace_error
	ace_init_error
	audit_trail
	emDbInfo_debug
	emDbInfo_info
	emDbObject_debug
	emDbObject_info
	security_alarm
Action messages	actmsg_debug
Alarm services	alarmsvc_debug
	alarmsvc_error
Annotation secretary	anno_debug
	anno_error
	anno_info
ASN.1	asn_debug
	asn_error
	asn_info
	asn_trace
Backing store	bs_error
	bsblob_error
CMIP	cmip_debug
	cmip_error
	cmip_info

TABLE 15-3 em_debug Message Types *(Continued)*

Category	Message Types
Connection manager	cmip_trace
	connectMgr_error
	connectMgr_impl
	connectMgr_info
Debugging port	dbgport_error
	dbgport_trace
Distributed alarm log manager	dalarm_debug
	dalarm_impl
Distributed transaction processing	XA_error
	XA_info
	XA_xactimpl_info
	xactimpl_error
	discrim_info
Discriminators	
Event management module	emm_debug
	emm_error
	emm_info
	emm_stat_info
	emm_trace
	sieve_stat_info
	sv_debug
	sv_error
	sv_info
	exception_info
Filtering	filt_debug
	filt_error
	filt_info
Geographical maps	map_debug
	map_error
	map_trace

TABLE 15-3 em_debug Message Types *(Continued)*

Category	Message Types
High-level PMI	hi_debug
	hi_error
	hi_info
	hi_trace
Initialization	init_debug
	init_error
Log management	evr2oc_error
	evr2oc_info
	log_error
	log_info
Message routing module	mrn_debug
	mrn_error
	mrn_info
	mrn_trace
MDR	mdr_debug
	mdr_error
	mdr_info
	mdr_trace
Nerve Center interface	nc_deb
	nc_error
	nc_event
	nc_miniger_debug
	nc_miniger_error
	nc_mosi_debug
	nc_mosi_error
	nc_mosi_trace
	nc_poll
	nc_poll_debug
	nc_poll_error
	nc_poll_info

TABLE 15-3 em_debug Message Types (*Continued*)

Category	Message Types
OAM	nc_state
	ncam_if_debug
	nce_at_debug
	nce_at_error
	nce_at_info
	nce_debug
	nce_error
	nce_info
	nce_poll_debug
	nce_poll_error
	nce_poll_info
	nce_poll_trace
	nce_snmp_error
	nce_trace
	mi_info
	minigreg_debug
	minigreg_error
	minigreg_info
	mosireg_error
	pm_deb
	pm_error
	pm_trace
	rcl_debug
	oam_debug
	oam_error
	oam_info
	oam_trace
	oammsg_debug
	oamnot_debug
	oamrmt_debug

TABLE 15-3 `em_debug` Message Types (Continued)

Category	Message Types
Object collections	<code>oamsvc_error</code>
	<code>oamsvc_info</code>
	<code>swap_debug</code>
	<code>coll_debug</code>
	<code>coll_error</code>
Scheduler	<code>coll_trace</code>
	<code>sched_debug</code>
	<code>sched_error</code>
	<code>sched_info</code>
Shutdown manager	<code>shutdown_debug</code>
Topology	<code>topo_debug</code>
	<code>topo_error</code>
	<code>topo_info</code>
	<code>topo_trace</code>

15.3 Avoiding Common Problems

Avoid some of the most common problems encountered with custom Solstice EM applications by:

- Verifying attribute and class names
- Creating automatically named managed object instances appropriately
- Testing that scopes and filters are supported

15.3.1 Verifying Attribute and Class Names

Whenever the MIS responds with an error message of the type `Unknown object class` or `Unknown attribute`, verify that the attribute or managed object class name in your application code matches exactly the name in the GDMO document.

All GDMO identifiers, including managed object class names and attribute names, are case-sensitive. Therefore, the case and the spelling of an attribute or managed object class name in your application code must match exactly the name in its GDMO document. A common mistake is to specify the `systemId` attribute (note the lower case `d`) as `systemID`.

If the MIS still responds with an error after you have verified an attribute or class name, verify that the correct version of the GDMO document is loaded into the MDR. For more information, refer to Section 15.1.5 “Reloading GDMO Documents” on page 15-4.

15.3.2 Creating Automatically Named Managed Object Instances Appropriately

If you want to create an automatically named managed object, you must call the `create_within` function of the `Image` class. An attempt to create an automatically named managed object by calling the `create` function of the `Image` class fails.

Check the name binding between a managed object class and its superior managed object class to determine if instances of the class are automatically named. The name binding is part of the GDMO specification of a managed object. If the name binding specifies `CREATE WITH-AUTOMATIC-INSTANCE-NAMING`, instances of the managed object class are automatically named. For example, instances of the `topoNode` and `logRecord` managed object classes are automatically named.

For information on name bindings, refer to Section 2.2.6.1 “Name Bindings” on page 2-15.

15.3.3 Testing That Scopes and Filters are Supported

Some MPAs and agents may not fully support scopes and filters. Find out whether the agents your application will manage support scopes and filters before implementing them in your management application.

To test if an MPA or agent supports scopes and filters, write a simple application that sends management requests with scopes and filters. If the requests fail, verify that the failure is not due to an error in your application. If the failure is not due to an error in your application, the agent or MPA does not support scopes or filters.

If your application will be used with an MPA or agent that does not support scopes or filters, select managed objects by performing multiple derivations. In these derivations, avoid specifying scopes or filters to select objects that are accessed via an agent or MPA that does not support scopes or filters.

For information on how to select managed objects by derivation, refer to Section 6.3.1 “Defining the Membership by Derivation” on page 6-3.

The code in CODE EXAMPLE 15-2 shows how to work with an MPA that does not support filters. In this example, a filtered request for `satellite` and `channel` object instances will fail. Instead, an `Album` instance is created that contains all instances of the `satellite` class and their contained `channel` instances. An instance of the `AlbumImage` class is then created to obtain each `channel` instance.

CODE EXAMPLE 15-2 Replacing a Scope and a Filter With Multiple Derivations

```
...
#include <pmi/hi.hh>                                // High Level PMI
#include <rw/cstring.h>                              // Rogue Wave RWCString
...
void get_satellites()
{
    satellites = Album("collection of all satellites");

    // Automatically add Image objects to the Album object
    // based on derivation rules
    //
    if (!satellites.set_prop(duTRACKMODE, duTRACK)) {
        cout << satellites.get_error_string() << endl;
        return;
    }

    // By default, an Image object is in the down state and read only.
    // Specify automatic activation of new Image objects
    // which are children of the derivation FDN
    //
    if (!satellites.set_prop(duAUTOIMAGE, duYES)) {
        cout << satellites.get_error_string() << endl;
        return;
    }

    // Set up the distinguished name to start the derivation
    RWCString derive_str;
    derive_str = "/systemId='";
    derive_str += server;
    derive_str += "'/LV(1)";
    derive_str += "/CMISFilter(item:equality:{objectClass,satellite})";

    cout << "Deriving satellites: " << derive_str.data() << endl;

    if (!satellites.set_derivation((char *) derive_str.data())) {
        cout << "Failed to set derivation string." << endl;
    }
}
```

CODE EXAMPLE 15-2 Replacing a Scope and a Filter With Multiple Derivations *(Continued)*

```
        cout << satellites.get_error_string() << endl;
        return;
    }

    // Get the objects from the MIS
    //
    if (!satellites.derive()) {
        cout << "Derivation failed." << endl;
        cout << satellites.get_error_string() << endl;
        return;
    }
...
}
...
void get_channels()
{
    channels = Album("collection of all channels");

    // Automatically add Image objects to the Album object based on
    // derivation rules.

    if (!channels.set_prop(duTRACKMODE, duTRACK)) {
        cout << channels.get_error_string() << endl;
        return;
    }

    // By default, an Image object is in the inactive state and read only.
    // Specify automatic activation of new Image objects
    // which are children of the derivation FDN

    if (!channels.set_prop(duAUTOIMAGE, duYES)) {
        cout << channels.get_error_string() << endl;
        return;
    }

    // Set up the distinguished name to start the derivation
    AlbumImage ai;
    for (ai = satellites.first_image(); ai; ai = ai.next_image()) {
        Album tmpch = Album("temporary holder for channels");
        Image im(ai);
        if (im.get_error_type() != PMI_SUCCESS) {
            cout << im.get_error_string() << endl;
            exit(8);
        }

        DU objname = im.get_objname();
```



```

        cout << endl;
        cout << "Distinguished Name: ";
        cout << objname.chp() << endl;
        RWCString derive_str;
        derive_str = objname.chp();
        derive_str += "/LV(1)";
        cout << "Deriving Channels: " << derive_str.data() << endl;
        if (!tmpch.set_derivation((char *) derive_str.data())) {
            cout << "Failed to set derivation string." << endl;
            cout << satellites.get_error_string() << endl;
            return;
        }

        if (!tmpch.derive(300)) {
            cout << "Derivation failed." << endl;
            return;
        }
        channels.include(tmpch);
    }
    ...
}
    ...

```

15.4 Example Troubleshooting Scenarios

This section explains how to apply troubleshooting techniques described in this chapter to the following scenarios:

- Failure to set an attribute value
- Failure to process event notifications

15.4.1 Failure to Set an Attribute Value

If an application attempts but fails to set an attribute value, debug the application by:

- Verifying GDMO and ASN.1 specifications
- Trapping errors in high-level PMI function calls
- Trapping programming logic errors
- Monitoring communications with the MIS

15.4.1.1 Verifying GDMO and ASN.1 Specifications

To verify that your application code is consistent with the GDMO and ASN.1 specifications of your object model:

- Verify that the case and the spelling of the attribute or managed object class name in your application code matches exactly the name in its GDMO specification.
- Check the managed object class definition to ensure that the managed object class you specified includes the attribute that you want to set.
- Verify that the attribute that you want to set is settable. Specifically, the property list in the `ATTRIBUTES` construct of the attribute's GDMO specification must include the operation you want to perform when setting the attribute. For more information, refer to Section C.4 "Package Template" on page C-7.
- Use the MIS Objects tool to set the attribute. If you succeed, make sure your application code matches the attributes you used in the MIS Objects tool.

15.4.1.2 Trapping Errors in High-Level PMI Function Calls

To trap errors in high-level PMI function calls:

- Use the overloaded not (!) operator on high-level PMI function calls before the attempt to set the attribute to verify that those calls succeeded.
- Call the `get_error_string` function of the `Image` class after the attempt to set the attribute. The error string returned may provide valuable information on the cause of the problem.

For more information on handling errors in high-level PMI function calls, refer to Chapter 4.

15.4.1.3 Trapping Programming Logic Errors

To verify that the program is executing the code that you expect it to execute:

- Set the `-g` flag when you compile the code, then run a debugger and step through the code one line at a time.
- Add print statements to verify that the program calls the function to set the attribute. You may find, for example, that the function is inside an `if` statement that is never called.

15.4.1.4 Monitoring Communications With the MIS

To verify that the MIS receives the set request and sends a successful response:

- Run `em_debug`, specifying `"on oam"` to print all object access messages that are sent to the MIS.
- Check the debug output for the set request. It is possible that the MIS never receives the request.
- Check the debug output for a response to the set request. The response may contain more useful information about why the request failed.

15.4.2 Failure to Process Notifications

If an application has registered to receive a type of event notification, but appears not to be acting on event notifications of that type, debug the application by:

- Verifying GDMO and ASN.1 specifications
- Trapping errors in high-level PMI function calls
- Trapping programming logic errors
- Monitoring communications with the MIS

15.4.2.1 Verifying GDMO and ASN.1 Specifications

To verify that your application code is consistent with the GDMO and ASN.1 specifications of your object model:

- Verify that the case and the spelling of the event type in your application code matches exactly the event type in its GDMO specification.
- Use the MIS Objects tool and watch the status line for event notifications of the type your application has registered to receive.

15.4.2.2 Trapping Errors in High-Level PMI Function Calls

To trap errors in high-level PMI function calls:

- Use the overloaded not (!) operator on high-level PMI function calls before the registration of the callback for the event type to verify that those calls succeeded.
- Verify that you are not excluding the event type in your application discriminator.
- Use the MIS Objects tool to check the `subsystemId="EM-MIS"/emApplicationId=yourApplicationId` and check the value for the discriminator attribute.

For more information on handling errors in high-level PMI function calls, refer to Chapter 4.

15.4.2.3 Trapping Programming Logic Errors

To verify that the program is executing the code that you expect it to execute:

- Verify that you have registered the correct callback function for the event type you want to handle.
- Add print statements to verify if your callbacks have been called.
- Set the `-g` flag when you compile the code, then run a debugger. Set breakpoints in the callback functions and check that the breakpoints are reached.

15.4.2.4 Monitoring Communication With the MIS

To verify that event notifications that your application has registered to receive are reaching the MIS:

- Run `em_debug`, specifying `"on not*"` to print out all notifications that are sent to the MIS.
- Check the `em_debug` output for event notifications of the type your application has registered to receive. It is possible that the event notifications of the type are never sent to the MIS.
- If the source of the event notification is an SNMP agent, turn on SNMP debugging, and monitor that trace to verify that the event notification was received by the SNMP MPA.

Integrating Applications With Solstice EM

Integrating applications with Solstice EM enables users to start your applications from the Solstice EM platform. Applications you develop will typically be used in conjunction with other Solstice EM components to provide a complete network management solution. Integrating applications with Solstice EM enhances the usability of your network management solution by providing users with the most convenient means to start applications. Integrating applications with Solstice EM also ensures that information required by the applications when they are started is always passed to them. Depending on the purpose of an application, you can integrate the application by adding it to a tools window, extending the Tools menu of a Solstice EM tool, or customizing the Network Views tool.

This chapter explains how to integrate applications with Solstice EM.

- Section 16.1 “Adding an Application to a Tools Window” on page 16-1
- Section 16.2 “Extending the Tools Menu of a Solstice EM Tool” on page 16-4
- Section 16.3 “Customizing the Network Views Tool” on page 16-6

16.1 Adding an Application to a Tools Window

Users of the Solstice EM platform start tools from a tools window. Each tools window groups together a number of related tools and provides for each tool an icon which, when clicked, starts the tool. Integrate an application with Solstice EM by adding the application to a tools window if the application is independent of other Solstice EM components in your network management solution.

Add an application to one of the following tools windows, depending on the purpose of the application:

- **Network Tools** if the application is used for network management
- **Administration** if the application is used for administration of your network management solution

To add an application to a tools window, edit one of the configuration files given in TABLE 16-1, depending on the window you want to add the application to.

TABLE 16-1 Configuration Files for Solstice EM Tools Windows

Window	Configuration File
Network Tools	/opt/SUNWconn/em/config/em_panel.cf
Administration	/opt/SUNWconn/em/config/em_admintool.cf

Note – The tools windows enable you to add applications to them interactively instead of by editing a configuration file. For information on how to add applications interactively to a tools window, refer to *Managing Your Network*.

Each configuration file for a tools window contains an entry for each application that you can start from the tools window. The format of an entry is as follows:

```
toolsWindow.glyphn: glyphFile
toolsWindow.labeln: labelText
toolsWindow.commandn: command
toolsWindow.responsen: response
toolsWindow.app_namen: applicationName
```

The variable parts of this format are explained in TABLE 16-2.

TABLE 16-2 Variable Parts in a Configuration File Entry for a Tools Window

<i>toolsWindow</i>	Specifies the tools window that the entry applies to. <i>toolsWindow</i> is one of the following keywords: <ul style="list-style-type: none"> • <i>em_panel</i> - Network Tools window • <i>em_admintool</i> - Administration window
<i>n</i>	An integer that specifies where in relation to other icons the icon for this application is displayed in the tools window. Icons in a tools window are displayed left to right and top to bottom.
<i>glyphFile</i>	The name of the file, including the full path, that contains icon that is displayed in the tools window. By default, Solstice EM glyph files are located in the /opt/SUNWconn/em/glyphs directory.

TABLE 16-2 Variable Parts in a Configuration File Entry for a Tools Window (Continued)

<i>labelText</i>	The text label displayed beneath the icon in the tools window. To create a multiline label, insert \n where you want a line break.
<i>command</i>	The command to start the application, including: <ul style="list-style-type: none">• The full path to the command• Any arguments that must be passed to the application when it is started
<i>response</i>	Specifies whether the tools window waits for a response from the application when a user starts the application. <i>response</i> is one of the following values: <ul style="list-style-type: none">• 0 - The tools window does not wait for a response. The icon for the application in the tools window remains grayed out and inactive for 5 seconds after a user starts the application.• 1 - The tools window waits for a response from the application to confirm that the application has started. The icon for the application in the tools window remains grayed out and inactive until the tools window receives a response.
<i>applicationName</i>	The name of the application as registered when the application was placed under security control. For information on how to place an application under security control, refer to <i>Managing Your Network</i> .

CODE EXAMPLE 16-1 shows an entry in the configuration file for the Network Tools window.

CODE EXAMPLE 16-1 Network Tools Window Configuration File Entry

```
em_panel.glyph4: /opt/SUNWconn/em/glyphs/em_datacollector.pm
em_panel.label4: Data\nCollections
em_panel.command4: /opt/SUNWconn/em/bin/em_datacollector -host EM_MIS
em_panel.response4: 1
em_panel.app_name4: em_datacollector
```

In this example, an application is added in the fourth position of the Network Tools window. The properties of this application are as follows:

- The icon that represents the application is contained in the file
/opt/SUNWconn/em/glyphs/em_datacollector.pm.
- The text label Data Collections is displayed beneath the icon. A line break is inserted after the word Data in this label.
- The command to start the application is
/opt/SUNWconn/em/bin/em_datacollector. The -host EM_MIS argument is passed to the application when it is started.
- The icon for this application remains grayed out and inactive until the Network Tools window receives a response from the application.
- The application was registered as em_datacollector when it was placed under security control.

16.2 Extending the Tools Menu of a Solstice EM Tool

Extending the Tools menu of a Solstice EM tool enables users to start your application by choosing a command from the Tools menu of a Solstice EM tool. Integrate an application by extending a Solstice EM tool if the purpose of the application is to provide additional features that the tool does not already provide, or if the application operates on a selection made in the tool.

For example, consider an application that processes alarms in a specialized manner. Any alarm that the application processes must be selected in the Alarms tool. To integrate this application with Solstice EM, the Alarms tool would be extended to add the command for starting the application to the Tools menu of the Alarms tool.

To extend the Tools menu of a Solstice EM tool, edit one of the configuration files given in TABLE 16-3, depending on the tool.

TABLE 16-3 Configuration Files for Solstice EM Tools

Solstice EM Tool	Configuration File
Network Views	/opt/SUNWconn/em/config/em_viewer.cf
Alarms	/opt/SUNWconn/em/config/em_alarmmgr_tp.cf
Event Logs	/opt/SUNWconn/em/config/em_logmgr_tp.cf
Log Entries	/opt/SUNWconn/em/config/em_logview_tp.cf

Note – Solstice EM tools enable you to extend their Tools menus interactively instead of by editing a configuration file. For information on how to extend the Tools menu of a Solstice EM tool interactively, refer to *Managing Your Network*.

Each configuration file for a Solstice EM tool contains an entry for each command on the Tools menu of the tool. To extend the Tools menu of a Solstice EM tool, add an entry to the configuration file for the tool. The format of an entry is as follows:

```
Application
{
  name: commandLabel
  path: command
  args: argList
}
```


Each entry starts with the keyword `Application`. The remainder of the entry is enclosed in braces.

The variable parts of a configuration file entry for a Solstice EM tool are explained in TABLE 16-4.

TABLE 16-4 Variable Parts in a Configuration File Entry for a Solstice EM Tool

<i>commandLabel</i>	The name of the command for starting the application as it appears on the Tools menu. If choosing this command opens another window, append an ellipsis to the command name.
<i>command</i>	The command to start the application, including the full path to the command.
<i>argList</i>	A list of arguments that must be passed to the application when the application is started. Each argument in <i>argList</i> is separated from the argument that precedes it by a space.

CODE EXAMPLE 16-2 shows an entry for extending the Tools menu in the configuration file for a Solstice EM tool.

CODE EXAMPLE 16-2 Configuration File Entry for Extending the Tools Menu

```
Application
{
  name: Alarms...
  path: /opt/SUNWconn/em/bin/em_alarmmgr
  args: -host EM_MIS
}
```

In this example, the Alarms command is added to the Tools menu of a Solstice EM tool. When a user chooses this command, the application that has the executable file `/opt/SUNWconn/em/bin/em_alarmmgr` is started. The argument `-host EM_MIS` is passed to the application when it is started. The ellipsis indicates that another window is opened when a user chooses this command.

16.3 Customizing the Network Views Tool

The actions that the Network Views tool provides depend on the topology type of the object selected in the Network Views window. Integrate an application by customizing the Network Views tool if the application is intended to process objects of a particular topology type. Depending on the purpose of the application, you can customize the Network Views tool by:

- Extending the Actions menu of the Network Views tool
- Setting the activation of a topology type

Note – The following subsections explain how to customize the Network Views tool by editing a configuration file. Solstice EM enables you to customize the Network Views tool interactively. For information on how to customize the Network Views tool interactively, refer to *Managing Your Network*.

16.3.1 Extending the Actions Menu of the Network Views Tool

Extending the Actions menu of the Network Views tool enables users to start your application by choosing a command from the Actions menu of the Network Views tool. The Actions menu of the Network Views tool provides additional custom commands that depend on the topology type of the object selected in the Network Views window. Integrate an application by extending the Actions menu of the Network Views tool if the application provides a specialized action for a particular topology type.

To extend the Actions menu of the Network Views tool, edit the `/opt/SUNWconn/em/config/em_viewer.cf` configuration file. This file contains an entry for each topology type to specify the custom commands on the Actions menu. The format of an entry that defines custom commands for a topology type is as follows:

```
Menu topoType
{
  "commandText1" command1
  .
  .
  .
  "commandTextN" commandN
}
```

Each entry starts with the keyword `Menu` followed by the topology type. Each command that you want to add to the Actions menu is specified on a separate line. The commands are enclosed in braces.

The variable parts of a configuration file entry that defines custom commands on the Actions menu for a topology type are explained in TABLE 16-5.

TABLE 16-5 Variable Parts in a Configuration File Entry for the Actions Menu

<i>topoType</i>	The topology type that the custom commands apply to.
<i>commandText1</i>	The name of the first custom command on the Actions menu. If choosing this command opens another window, append an ellipsis to the command name. <i>commandText1</i> is enclosed in double quotes.
<i>command1</i>	The command to start the application associated with the first custom command, including: <ul style="list-style-type: none">• The full path to the command• Any arguments that must be passed to the application when it is started
<i>commandTextN</i>	The name of the Nth custom command on the Actions menu. If choosing this command opens another window, append an ellipsis to the command name. <i>commandTextN</i> is enclosed in double quotes
<i>commandN</i>	The command to start the application associated with the Nth custom command, including: <ul style="list-style-type: none">• The full path to the command• Any arguments that must be passed to the application when it is started

CODE EXAMPLE 16-3 shows an entry for extending the Actions menu of the Network Views tool for a topology type.

CODE EXAMPLE 16-3 Extending the Actions Menu of the Network Views Tool

```
Menu Satellite
{
"Display View"      EM_GOTOVIEW
"Alarms..."       /opt/SUNWconn/em/bin/em_alarmmgr -host EM_MIS -id EM_UNIQUE_ID
"Dish Watcher..." /opt/SUNWconn/em/bin/graphics
"Properties..."    /opt/SUNWconn/em/bin/em_oct -host EM_MIS -id EM_UNIQUE_ID
}
```

In this example, the following commands are added to the Actions menu of the Network Views tool for the Satellite topology type:

- **Display View.** When a user chooses this command, the Network Views tool changes the current view to the associated object.
- **Alarms.** When a user chooses this command, the application that has the executable file `/opt/SUNWconn/em/bin/em_alarmmgr` is started. The arguments `-host EM_MIS` and `-id EM_UNIQUE_ID` are passed to the application when it is started. The ellipsis indicates that another window is opened when a user chooses this command.
- **Dish Watcher.** When a user chooses this command, the application that has the executable file `/opt/SUNWconn/em/bin/graphics` is started. No arguments are passed to the application when it is started. The ellipsis indicates that another window is opened when a user chooses this command.
- **Properties.** When a user chooses this command, the application that has the executable file `/opt/SUNWconn/em/bin/em_oct` is started. The arguments `-host EM_MIS` and `-id EM_UNIQUE_ID` are passed to the application when it is started. The ellipsis indicates that another window is opened when a user chooses this command.

Note – `EM_GOTOVIEW` is a Network Views command macro. Assign this command macro only to topology types that are a view (for example, Container or Monitor types). For more information, refer to *Managing Your Network*.

16.3.2 Setting the Activation of a Topology Type

Setting the activation of a topology type specifies the action carried out when a topology node of the type in the Network Tools window is double clicked. Integrate an application by setting the activation of a topology type if the application performs the default action for the topology type. The application is run when a topology node of the type in the Network Tools window is double clicked.

To set the activation of a topology type, edit the `/opt/SUNWconn/em/config/em_viewer.cf` configuration file. This file contains an entry that sets the activations of all topology types that are displayed in the Network Tools window. The format of this entry is as follows:

```
Activations
{
  "topoType1"  command1
  .
  .
  .
  "topoTypeN"  commandN
}
```

This entry starts with the keyword `Activations`. The activation setting for each topology type is specified on a separate line. The activation settings are enclosed in braces.

The variable parts of the configuration file entry that sets activations of topology types are explained in TABLE 16-6.

TABLE 16-6 Variable Parts of the Configuration File Entry That Sets Activations

<i>topoType1</i>	The first topology type that you want to set the activation of. <i>topoType1</i> is enclosed in double quotes.
<i>command1</i>	The command that specifies the action carried out when a topology node of type <i>topoType1</i> is double clicked. You must include: <ul style="list-style-type: none">• The full path to the command• Any arguments that must be passed to the application when it is started
<i>topoTypeN</i>	The Nth topology type that you want to set the activation of. <i>topoTypeN</i> is enclosed in double quotes.
<i>commandN</i>	The command that specifies the action carried out when a topology node of type <i>topoTypeN</i> is double clicked. You must include: <ul style="list-style-type: none">• The full path to the command• Any arguments that must be passed to the application when it is started

CODE EXAMPLE 16-4 shows an example of the configuration file entry that defines activations of topology types.

CODE EXAMPLE 16-4 Setting the Activations of Topology Types

```
Activations
{
  "Array"      EM_GOTOVIEW
  "Bridge"     /opt/SUNWconn/em/bin/em_alarmmgr -host EM_MIS -id EM_UNIQUE_ID
  "Dish"       /opt/SUNWconn/em/bin/graphics
}
```

In this example, activations of topology types are set as follows:

- When a topology node of type `Array` is double clicked, the Network Views tool changes the current view to the associated object.
- When a topology node of type `Bridge` is double clicked, the application that has the executable file `/opt/SUNWconn/em/bin/em_alarmmgr` is started. The arguments `-host EM_MIS` and `-id EM_UNIQUE_ID` are passed to the application when it is started.
- When a topology node of type `Dish` is double clicked, the application that has the executable file `/opt/SUNWconn/em/bin/graphics` is started. No arguments are passed to the application when it is started.

Note – `EM_GOTOVIEW` is a Network Views command macro. Assign this command macro only to topology types that are a view (for example, `Container` or `Monitor` types). For more information, refer to *Managing Your Network*.

Writing RPC Agents for Solstice EM

Solstice Enterprise Manager (Solstice EM) supports agents written to the Site/SunNet/Domain Manager (SNM) interfaces and libraries. SNM agents can communicate with the Solstice EM MIS via the Solstice EM MIS RPC interface. The Solstice EM MIS RPC interface translates from SNM ONC RPC to the Solstice EM PMI.

This chapter explains how to write RPC agents for Solstice EM.

- Section 17.1 “Manager-Agent Model” on page 17-1
- Section 17.2 “Types of Agents” on page 17-2
- Section 17.3 “Steps for Writing an Agent” on page 17-3
- Section 17.4 “Solstice EM Integration” on page 17-4

17.1 Manager-Agent Model

The SNM design is based on the manager/agent model in the Open Systems Interconnection (OSI) management framework. The manager is a process started by the user (for example, the Solstice EM MIS). The agent is a process that collects data from the managed resource and reports it to the manager.

The Manager/Agent Services libraries provide the management infrastructure and handle the communication services. The agent and manager need not be concerned with the underlying networking involved in their communication. The agent process is concerned only with collecting data from the managed resource. The manager and agent processes make use of the Services through Application Programming Interfaces (APIs).

Another aspect of the manager/agent model involves the definition of management data. Open management standards—for example, OSI and the Simple Network Management Protocol (SNMP)—specify that agents abstract the properties (or attributes) of managed resources into data items (for example, “how busy a CPU is”

becomes a value between 0 and 100). In SNMP, the attributes for a managed resource are described in the agent *schema*. The agent is able to respond to the manager's request, because both use the same data definitions for the managed resource.

17.2 Types of Agents

All SNMP agents communicate with the manager in the manner just described. Agent types differ in the relationship with their respective managed resources.

Agents can directly or indirectly access managed resources. Most of the SNMP agents provided with Solstice EM manage resources on the Sun workstations where they are installed. For example, the *hostmem* agent uses the same mechanism as `netstat -m` to get memory utilization data.

The second type of agent provides the ability to manage objects that are not directly accessible. Such agents are called *proxy agents*. Proxy agents run on Sun workstations, called *proxy systems*, and use protocol translation mechanisms to provide the necessary access to the managed resources. The proxy system may be the workstation on which the Solstice EM MIS is running or another workstation on the network.

The ping proxy agent provides the ability to test the reachability of Internet Protocol (IP) devices, translating manager requests into Internet Control Message Protocol (ICMP) echo requests. Similarly, the *hostperf* proxy agent uses the *rstat* protocol to gather host statistics.

Proxy agents provide a mechanism allowing SNMP to extend into virtually any domain. The Simple Network Management Protocol (SNMP) proxy agent provides the ability to manage any device that supports SNMP, the widely accepted standard management protocol for the TCP/IP world.

17.3 Steps for Writing an Agent

Before an agent is written, access to the managed resource must exist (that is, code must be written to get the required management data).

Note – The prefixes `NETMGT`, `Netmgt`, and `netmgt` are reserved for network management functions.

From a high-level viewpoint, the steps involved in implementing an agent are as follows:

- Assigning a name to each discrete management data item—*attribute*. For instance, if the input packet count is an attribute, an appropriate name is `ipkts`.
- Determining the data type for each attribute. In the example, `ipkts` is an integer.
- Using the attribute information to form the agent schema file, which will be specific to the particular agent.

Note – The conversion of SNM schema files into GDMO files for use with the Solstice EM MIS requires that only alphanumeric characters be used for names. Therefore, use special characters with caution.

- Expanding the original code written for accessing the managed resource to incorporate the agent schema definitions.
- Writing the code that uses the SNM Agent Services library. This includes code for agent initialization, request handling, and error reporting.
- Incorporating any agent-specific error messages into the agent schema file.
- Testing and integrating the completed agent code and schema file with Solstice EM.

17.4 Solstice EM Integration

Once you are satisfied your agent is performing correctly, install it where you want it to run on the systems, then integrate it with the Solstice EM MIS database.

17.4.1 Installing the Agent

For *each system* where you want your agent to run,

- Copy your agent to the directory where the other SNM agents are installed. If no SNM agents have been installed on the system, first run the SNM utility `getagents`.
- Add the following entry to `snm.conf`:

```
na.my-agent-name0
```

This will set your agent security level on this host to zero (no security checking). Optionally, you can set your agent to any value between 1 and 5.

- Add an entry for your agent to `/etc/inetd.conf` to allow `inetd` to automatically start your agent when a manager sends a request to your agent. Here's a summary of the `inetd.conf(5)` file format.

```
na.agent-name/10 tli rpc/udp wait root agent program absolute pathname agent-  
name arguments
```

For example, the entry for the `iproutes` agent (on a Solaris 2.x machine) is:

```
na.hostmem/10 tli rpc/udp wait root  
/opt/SUNWconn/snm/agents/na.iproutes na.iproutes
```

- Force `inetd` to reread its configuration file by sending it a `SIGHUP` signal:

```
hostname% kill -HUP inetd's process ID
```

17.4.2 Updating the Solstice EM MIS Database

Solstice EM uses GDMO descriptions, rather than the schema files used by SNM products, to represent the managed object classes available to the management database. Thus, the Solstice EM Schema compiler (`em_snm2gdm`) is used as the first step in converting the SNM schema files into GDMO descriptions used in Solstice EM.

To translate an SNM schema file, run the Schema compiler with the file to be translated as an argument (*filename*).

```
hostname% em_snm2gdm < filename
```

The output of the translation is a GDMO document, with a name of the form *filename.gdm*, and an ASN.1 document, with a name of the form *filename.asn1*. You must then pass the GDMO file through the GDMO compiler.

Solstice EM C++ Source Code Examples

Solstice EM is supplied with a number of source code examples that show how to use the Solstice EM C++ application programming interfaces (APIs) to develop network management applications. The examples are contained in subdirectories of the `/opt/SUNWconn/em/src` directory. For detailed information about an example, refer to the `README` file supplied with the example.

This appendix introduces the C++ source code examples supplied with Solstice EM.

- Section A.1 “Guidelines for Compiling the Examples” on page A-1
- Section A.2 “Satellite Example” on page A-2
- Section A.3 “High-Level PMI Examples” on page A-3
- Section A.4 “Scenario Examples” on page A-8
- Section A.5 “Security Examples” on page A-9
- Section A.6 “Low-Level PMI Examples” on page A-10
- Section A.7 “Object Modeling Example” on page A-11
- Section A.8 “Object Development Examples” on page A-11
- Section A.9 “Miscellaneous Examples” on page A-12

A.1 Guidelines for Compiling the Examples

To simplify compiling the examples, a `Makefile` is supplied with each example.

Note – Before trying to compile an example, make sure that the output files are written to a directory you have write access to. The file access permissions of the directories under `/opt/SUNWconn` allow only the root user to write these directories.

If you want to compile an example by using the `CC` command, refer to Chapter 14 for guidelines on how to compile applications developed with the Solstice EM C++ APIs.

To compile the examples, you must use version 5.2 of the Sun™ Workshop C++ compiler. To verify the compiler version, type the following command:

```
prompt% CC -V
```

A.2 Satellite Example

The satellite example illustrates applications to manage a number of satellites by:

- Monitoring the traffic on the satellites
- Controlling the geographical coordinates of the satellites
- Monitoring alarms and events
- Changing communication parameters of the satellites

The example is contained in subdirectories of the `/opt/SUNWconn/em/src/satellite` directory. The files in each subdirectory illustrate a particular aspect of the application development process as shown in TABLE A-1.

TABLE A-1 Subdirectories of the Satellite Example Directory

Subdirectory	Illustrates
mgmt_info	Developing the object model of the satellites
platform	Enabling an application to access managed objects
actions	Retrieving data from the metadata repository
objects	Performing operations on managed objects
collection	Performing management operations on object collections
graphics	Handling events in a GUI application
security	Making applications and managed objects secure
simulate	Simulating events
discover	Handling network topology

TABLE A-1 Subdirectories of the Satellite Example Directory *(Continued)*

Subdirectory	Illustrates
complex	Encoding and decoding complex data types
performance	Optimizing performance by saving memory and speeding up event processing
low_level	Optimizing performance by using the low-level portable management interface (PMI) API

A.3 High-Level PMI Examples

The `/opt/SUNWconn/em/src/pmi_hi` directory contains examples that show how to use the high-level PMI. The examples show how to accomplish tasks related to several different aspects of network management as described in the following subsections.

A.3.1 Managed Object Examples

TABLE A-2 lists examples showing how to perform management operations on managed objects. The examples are contained in the `/opt/SUNWconn/em/src/pmi_hi` directory.

TABLE A-2 Managed Object Examples for the High-Level PMI

File	Description
<code>get.cc</code>	<p>Synchronously gets the value of the specified attribute for the specified managed object. If no attribute is specified, the program gets the values of all attributes for the specified managed object. If no arguments are specified, the program gets the values of all attributes for the topology database <code>topoType</code> object named <code>Host</code>. To start the program after compiling it, type:</p> <pre>prompt% get [fdn moc [attribute]]</pre> <p>Where:</p> <ul style="list-style-type: none"> <code>fdn</code> is the fully distinguished name (FDN) of the managed object. <code>moc</code> is the managed object class of the object. <code>attribute</code> is the attribute you want to get.
<code>get_asyn.cc</code>	<p>Asynchronously gets an attribute value of a managed object. This example is an asynchronous version of <code>get.cc</code>.</p>

TABLE A-2 Managed Object Examples for the High-Level PMI (*Continued*)

File	Description
<code>set.cc</code>	<p>Synchronously sets the value of the specified attribute for the specified managed object. If no arguments are specified, the program sets the poll rate of the NerveCenter PollRate Fast object to 777. To start the program after compiling it, type:</p> <div><pre>prompt% set <i>fdn moc attribute value</i></pre></div> <p>Where:</p> <ul style="list-style-type: none">• <i>fdn</i> is the FDN of the managed object.• <i>moc</i> is the managed object class of the object.• <i>attribute</i> is the attribute you want to set.• <i>value</i> is the value you want to set the attribute to.
<code>set_asyn.cc</code>	<p>Asynchronously sets an attribute value of a managed object. This example is an asynchronous version of <code>set.cc</code>.</p>
<code>image_boot.cc</code>	<p>Loads information on the specified attributes of the specified managed object. If no attributes are specified, information on all attributes of the managed object is loaded. To start the program after compiling it, type:</p> <div><pre>prompt% image_boot -d <i>fdn</i> [-a <i>attr1</i> -a <i>attr2</i> ... -a <i>attrN</i>]</pre></div> <p>Where:</p> <ul style="list-style-type: none">• <i>fdn</i> is the FDN of the managed object.• <i>attri</i> are the specified attribute names. If no attributes are specified, all attributes are obtained.
<code>object_count.cc</code>	<p>Counts all local objects, stores FDN entries in a hash table, and searches the entries in the hash table.</p>

A.3.2 Object Collection Examples

TABLE A-3 lists examples showing how to perform management operations on object collections. The examples are contained in the `/opt/SUNWconn/em/src/pmi_hi` directory.

TABLE A-3 Object Collection Examples for the High-Level PMI

File	Description
<code>album.cc</code>	Synchronously derives an object collection. For each object in the collection, the example program prints the FDN of the object, and for each attribute its name and value. To start the program after compiling it, type: <div><pre>prompt% album [-h hostName]</pre></div> Where <i>hostName</i> is the name of the host where the MIS is running. If it is not specified, <code>localhost</code> is assumed.
<code>album_asyn.cc</code>	Asynchronously derives an object collection. This example is an asynchronous version of <code>album.cc</code> .
<code>album_logObj.cc</code>	Derives a collection of all <code>log</code> objects created under the host where the MIS is running. If a new <code>log</code> object is created, an <code>Image</code> instance representing it is added to the object collection.
<code>album_wait.cc</code>	Derives an object collection the membership of which is maintained automatically.
<code>derive_rpc.cc</code>	Derives an object collection for RPC groups, for the specified host and MIS.
<code>derive_snmp.cc</code>	Derives an object collection for SNMP MIB II groups, for the specified host and MIS.

A.3.3 Event Handling Examples

TABLE A-4 lists examples showing how to handle events. The examples are contained in the `/opt/SUNWconn/em/src/pmi_hi` directory.

TABLE A-4 Event Handling Examples for the High-Level PMI

File	Description
<code>efd.cc</code>	Creates an event forwarding discriminator (EFD) and sets up an MIS to forward events to another MIS.
<code>event_send1.cc</code>	Sends an event notification containing the event name, event information, and a default time stamp.

TABLE A-4 Event Handling Examples for the High-Level PMI (*Continued*)

File	Description
event_send2.cc	Sends an event notification containing the event name, event information, and a custom time stamp.
event_send3.cc	Sends an event notification containing the event name, event information, and a custom time stamp in ASN.1 notation.
event.cc	Listens for all events and prints the contents of any event it receives.
event_app1.cc	Listens for specific event types and prints the contents of any event it receives.
event_app2.cc	Listens for events of all types that happen to instances of the specified managed object classes. To start the program after compiling it, type: <div data-bbox="612 644 1255 687" style="border: 1px solid black; padding: 2px; margin: 5px 0;"> prompt% event_app2 <i>moc1</i> <i>moc2</i> ... <i>mocN</i> </div> Where <i>moci</i> are the specified managed object classes.
event_create_logObj.cc	Listens for and prints log object creation events.
plat_get_conn_fd.cc	Gets and prints the file descriptor for an MIS connection.

A.3.4 Log Record Handling Examples

TABLE A-5 lists examples showing how to handle log records. The examples are contained in the `/opt/SUNWconn/em/src/pmi_hi` directory

TABLE A-5 Log Record Handling Examples for the High-Level PMI

File	Description
create.cc	Synchronously creates a log object.
create_asyn.cc	Asynchronously creates a log object. This example is an asynchronous version of <code>create.cc</code> .
delete.cc	Deletes the specified log object.
delete_asyn.cc	Asynchronously deletes the specified log object. This example is an asynchronous version of <code>delete.cc</code> .

A.3.5 Network Topology Examples

TABLE A-6 lists examples showing how to read and modify topology node data. The examples are contained in the `/opt/SUNWconn/em/src/pmi_hi/topo_user_data` directory.

TABLE A-6 Network Topology Examples for the High-Level PMI

File	Description
<code>get_topoNodeUserData.cc</code>	Gets the value of a topology node attribute in compound and scalar format.
<code>set_topoNodeUserData.cc</code>	Sets the value of a topology node attribute.

A.3.6 FDN Translation Examples

TABLE A-5 lists examples showing how to translate FDNs of managed objects into nicknames. The examples are contained in the `/opt/SUNWconn/em/src/pmi_hi/dn_translation` directory.

TABLE A-7 FDN Translation Examples for the High-Level PMI

File	Description
<code>fdn_translate.cc</code>	Gets partial nicknames of Image instances that represent managed objects.
<code>get_nickname_NNsrvr.cc</code>	Retrieves the FDN translation of a managed object and sets the nickname of an Image instance to the translated name.
<code>set_nickname_NNsrvr.cc</code>	Sets the nickname of an Image instance and defines this nickname in the nickname server.

A.3.7 Graphical Application Examples

The `/opt/SUNWconn/em/src/pmi_hi/cmipconfig` directory contains an example of a graphical application for setting up MIS-to-MIS communication. For information on the components of this example, refer to the `/opt/SUNWconn/em/src/pmi_hi/cmipconfig/README` file.

A.3.8 MDR Action Examples

The `/opt/SUNWconn/em/src/pmi_hi/mdr_action.cc` file contains an example of how to use actions to retrieve data from the metadata repository (MDR).

A.3.9 Encoding and Decoding Examples

The `/opt/SUNWconn/em/src/pmi_hi/morf_topoNode.cc` file contains an example of how to use the `Morf` class to split a compound data attribute value into scalar data values.

A.4 Scenario Examples

The scenario examples show how to use several components of Solstice EM together to develop a complete network management solution. The examples are contained in subdirectories of the `/opt/SUNWconn/em/src/scenario` directory. The scenarios are covered by the files in one or more of the subdirectories as follows:

TABLE A-8 Scenario Examples

Scenario	Subdirectory
Adding and managing a new class of device	example1
	example2
Using PMI client applications to present information	example3
	example4
	example5
Sharing information among PMI client applications	example6
Using multiple MISs	example7
	example8
	example9

For more information on each example, refer to the `README` file in the subdirectory that contains the example.

A.5 Security Examples

The security examples show how to use the access control mechanisms of Solstice EM to make applications and managed objects secure.

A.5.1 Access Control API Examples

TABLE A-9 lists examples showing how to use the access control API to make applications and managed objects secure. The examples are contained in the `/opt/SUNWconn/em/src/ac_api` directory.

TABLE A-9 Access Control API Examples

File	Description
<code>app_feature_list.cc</code>	Lists applications and their features.
<code>create_em_user.cc</code>	Configures access control for applications.
<code>create_rule.cc</code>	Creates a security rule.
<code>create_target.cc</code>	Configures access control for managed objects.
<code>list_security_defaults.cc</code>	Lists security defaults.

A.5.2 Access Control Engine API Examples

TABLE A-9 lists examples showing how to use the access control engine API to make a management protocol adapter (MPA) or protocol driver module (PDM) secure. The examples are contained in the `/opt/SUNWconn/em/src/mpa_pdm/src` directory.

TABLE 17-1 Access Control Engine API Examples

File	Description
<code>mpapdm_ace.cc</code>	Implements the <code>MpapdmAuxServer</code> class.
<code>msgio.cc</code>	Checks access control and cleans up access control objects.
<code>req_mngt.cc</code>	Checks access control on pending MPA PDM requests.
<code>samp_main.cc</code>	Initializes access control objects.

A.5.3 Password Request Example

The `/opt/SUNWconn/em/src/pmi_hi/passwd_dialog/dialog_box.cc` file contains an example of how to create a dialog box that requests a password from a user to before allowing access to an application.

A.5.4 Application-Feature-Level Example

The `/opt/SUNWconn/em/src/pmi_hi/access_feature/access_feature_level.cc` file contains an example of how to use the high-level PMI to add feature-level access control to an application.

A.6 Low-Level PMI Examples

TABLE A-10 lists examples showing how to use the low-level PMI. The examples are in the `/opt/SUNWconn/em/src/pmi_low` directory.

TABLE A-10 Low-Level PMI Example Programs

File	Description
<code>album_low.cc</code>	Sends a scoped <code>GetReq</code> command. The high-level PMI equivalent is <code>Album::derive</code> with scoping.
<code>boot_low.cc</code>	Sends a <code>GetReq</code> command with a selected attribute list. The high-level PMI equivalent is <code>Image::boot(attrlist)</code> .
<code>create_low.cc</code>	Sends a <code>CreateReq</code> command with a selected attribute list. The high-level PMI equivalent is <code>Image::set</code> calls followed by <code>Image::create</code> .
<code>delete_low.cc</code>	Sends a <code>DeleteReq</code> command. The high-level PMI equivalent is <code>Image::destroy</code> .
<code>event_gen_low.cc</code>	Sends events directly to the event manager in the MIS.
<code>filter_low.cc</code>	Sends a scoped, filtered <code>GetReq</code> command. The high level PMI equivalent is <code>Album::derive</code> with scoping and filtering.
<code>set_low.cc</code>	Sends a <code>SetReq</code> command with a selected attribute list. The high-level PMI equivalent is <code>Image::set</code> calls followed by <code>Image::store</code> .
<code>simple_low.cc</code>	Sends a simple <code>GetReq</code> command for the root object. No attributes are retrieved. There is no high level PMI equivalent.

A.7 Object Modeling Example

The `/opt/SUNWconn/em/src/gdmo_sample` directory contains the GDMO definition and ASN.1 module specification of an example managed object class.

A.8 Object Development Examples

The object development examples show how to use object development tools (ODT) to develop object behaviors. Each example is contained in a subdirectory of the `/opt/SUNWconn/src/odt` directory as listed in TABLE A-11.

TABLE A-11 Object Development Examples

Subdirectory	Description
<code>cellSample</code>	Defines a set of complex intra-object behaviors. This example monitors an object. If the behavior of the monitored object changes, the example changes the behavior of the monitored object's neighboring objects.
<code>chai</code>	Monitors an attribute named <code>chaiReady</code> to determine if there is any chai tea ready to drink. If not, the example sends an action named <code>brewChai</code> to make more chai tea.
<code>demoPing</code>	Defines behavior of a native agent.
<code>demoregistry</code>	Provides an MIS client function to operate as a remote agent. This example demonstrates how to register an application, similar to a licensing facility.
<code>demoServer</code>	Provides an MIS server function to operate as a remote agent. This example provides support required for the <code>demoregistry</code> and <code>diskInfo</code> examples.
<code>diskInfo</code>	Demonstrates behavior to get information from a process that is outside the MIS.

A.9 Miscellaneous Examples

The examples listed in TABLE A-12 are in subdirectories of the `/opt/SUNWconn/em/src` directory.

TABLE A-12 Miscellaneous API Examples

Subdirectory	Illustrates
<code>app_api</code>	Using the application-to-application API
<code>grapher_api</code>	Using the grapher API
<code>topo_api</code>	Using the topology API
<code>viewer_api</code>	Using the viewer API
<code>mpa_pdm</code>	Developing an MPA or PDM
<code>nci</code>	Sending Nerve Center requests
<code>nn</code>	Getting and setting nicknames
<code>objop</code>	Creating and modifying managed objects by using the <code>em_objop</code> utility
<code>odt</code>	Developing object behaviors
<code>overload</code>	Using the overload control mechanism of the CMIP MPA
<code>sql</code>	Generating reports by using the structured query language (SQL)

Standards Reference and Further Reading

B.1 Standards Reference

Solstice EM is based on the specifications and standards listed in the following subsections.

Telecommunications Management Network (TMN)

- M.3000 *Overview of TMN Recommendations*
- M.3010 *Principles of a Telecommunications Management Network*
- M.3020 *TMN Interface Specification Methodology*
- M.3100 *Generic Network Information Model*
- M.3180 *Catalogue of TMN Management Information*
- M.3200 *TMN Management Services: Overview*
- M.3300 *TMN Management Capabilities*
- M.3400 *TMN Management Functions*

OSI Model and Notation

- ITU-T X.208/ISO-8824 *Specification of Abstract Syntax Notation One (ASN.1)*
- ITU-T X.209/ISO-8825 *Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1)*

OSI Structure of Management Information

- ITU-T X.700/ISO-7498 *Management Framework*
- ITU-T X.701/ISO-10040 *System Management Overview*
- ITU-T X.710/ISO-9595 *Common Management Information Services (CMISE)*
- ITU-T X.711/ISO-9596-1 *Common Management Information Protocol Specification*
- ITU-T X.720/ISO-10165-1 *Management Information Model*
- ITU-T X.721/ISO-10165-2 *Definition of Management Information*
- ITU-T X.722/ISO-10165-4 *Guidelines for the Definition of Managed Objects (GDMO)*

OSI Service Elements

- ITU-T X.219/ISO-9072-1 *Remote Operations: Model, Notation and Service Definition*
- ITU-T X.229/ISO-9072-2 *Remote Operations: Protocol Specification*
- ITU-T X.217/ISO-8649 *Service Definition for the Association Control Service Element*
- ITU-T X.227/ISO-8650 *Connection-Oriented Protocol Specification for the Association Control Service Element*

Systems Management Functions

- X.730/ ISO 10164-1 *Object Management Function*
- X.731/ ISO 10164-2 *State management Function*
- X.732/ ISO 10164-3 *Attributes for Representing Relationships*
- X.733/ ISO 10164-4 *Alarm Reporting Function*
- X.734/ ISO 10164-5 *Event Report Management Function*
- X.735/ ISO 10164-6 *Log Control Function*
- X.736/ ISO 10164-7 *Security Alarm Reporting Function*
- X.737/ ISO 10164-14 *Confidence and Diagnostic Test Categories*
- X.738/ ISO 10164-13 *Summarization Function*
- X.739/ ISO 10164-11 *Metric Objects and Attributes*
- X.740/ ISO 10164-8 *Security Audit Trail Function*
- X.741/ ISO 10164-9 *Objects and Attributes for Access Control*
- X.742/ ISO 10164-10 *Usage Metering Function for Accounting Purposes*
- X.745/ ISO 10164-12 *Test Management Function*
- X.746/ ISO 10164-15 *Scheduling Function*
- X.750/ ISO 10164-16 *Management Knowledge Management Function*
- X.751/ ISO 10164-17 *Changeover Function*

B.2 Terminology References

The ISO specifications provide precise technical definitions for a number of terms used through out this document.

TABLE B-1 ISO Specifications for Terminology Definitions

Term	Defined In
access control	X.741 ISO/IEC 10164-9: 1995(E)
action	X.720 ISO/IEC 10165-1
agent	X.701 ISO/IEC 10040
agent role	X.701 ISO/IEC 10040
Application Entity Title (AE-Title)	ISO 7498-3
Application Program Title (AP-Title)	ISO 7498-3
Association Control Service Element (ACSE) protocol	ITU-T X.227/ISO-8650
Association Control Service Element (ACSE) service	ITU-T X.217/ISO-8649
attribute	X.710 ISO/IEC 9595
attribute identifier	X.720 ISO 10165-1
attribute type	X.720 ISO 10165-1
attribute value assertion	X.720 ISO 10165-1
constructed encoding	X.209 ISO 8825
containment	X.720 ISO 10165-1
destination (see destination attribute syntax)	X.734 ISO 10164-5
discriminator	X.734 ISO 10164-5
distinguished name	X.720 ISO 10165-1
end-of-contents octets	X.209 ISO 8825
event forwarding discriminator (see eventForwardingDiscriminator managed object class syntax)	X.734 ISO 10164-5
event report management function	X.734 ISO 10164-5
filter (see CMISfilter syntax)	X.711 ISO/IEC 9596-1
functional unit	X.710 ISO/IEC 9595
identifier octets	X.209 ISO 8825

TABLE B-1 ISO Specifications for Terminology Definitions *(Continued)*

Term	Defined In
length octets	X.209 ISO 8825
local distinguished name (see ObjectInstance syntax)	X.711 ISO/IEC 9596-1
managed object	ISO 7498-4
managed object class	X.701 ISO/IEC 10040
management information	X.701 ISO/IEC 10040
managed information base	ISO 7498-4
manager	X.701 ISO/IEC 10040
manager role	X.701 ISO/IEC 10040
naming binding	X.720 ISO 10165-1
naming tree	X.720 ISO 10165-1
notification	X.701 ISO/IEC 10040
notification type	X.701 ISO/IEC 10040
open system	X.200 ISO 7498
primitive encoding	X.209 ISO 8825
relative distinguished name	X.720 ISO 10165-1
Remote Operations Service Element (ROSE) protocol	ITU-T X.229/ISO-9072-2
Remote Operations Service Element (ROSE) service	ITU-T X.219/ISO-9072-1
systems management	X.200 ISO 7498
top managed object class	X.721 ISO/IEC 10165-2

B.3 Further Reading

The following books are useful for more detailed information about OSI networking concepts and principles:

- *Computer Networks (Second Edition)* by Andrew S. Tanenbaum (Prentice-Hall International Editions, 1988)
- *OSI A Model for Computer Communications Standards* by Uyles Black (Prentice-Hall, 1991)
- *Network Management Standards (The OSI, SNMP and CMOL Protocols)* by Uyles Black (McGraw-Hill on Computer Communications, 1992)
- *SNMP, SNMPv2, and CMIP: The Practical Guide to Network-Management Standards* by William Stallings (Addison-Wesley, 1993)
- *Telecommunications Network Management into the 21st Century: Techniques, Standards, Technologies and Applications* edited by Salah Aidarous and Thomas Plevyak (IEEE Press, 1994)
- *GDMO Object Modelling and Definition for Network Management* by Baha Hebrawi (Technology Appraisals, 1995)

GDMO Templates

This appendix describes the templates defined by ITU-T X.722/ISO-10165-4 *Guidelines for the Definition of Managed Objects (GDMO)*.

- Section C.1 “Conventions Used in the Definitions” on page C-1
- Section C.2 “Managed Object Class Template” on page C-2
- Section C.3 “Name Binding Template” on page C-4
- Section C.4 “Package Template” on page C-7
- Section C.5 “Attribute Template” on page C-11
- Section C.6 “Action Template” on page C-13
- Section C.7 “Notification Template” on page C-15
- Section C.8 “Parameter Template” on page C-18
- Section C.9 “Attribute Group Template” on page C-20
- Section C.10 “Behaviour Template” on page C-22

C.1 Conventions Used in the Definitions

In the template definitions:

- Capitalized words are GDMO keywords.
- Words in angle brackets represent information that is filled in when a template is instantiated.
- An item in square brackets represents an optional value.
- A closing square bracket followed by an asterisk indicates that the item enclosed in brackets may be repeated zero or more times.

Note – Some items are references to an instance of use of another GDMO template. If an item is defined in a different GDMO document, the item must be specified as a fully-qualified name, that is: <GDMO-Document-Name> : <label>.

C.2 Managed Object Class Template

The Managed Object Class template references all other templates to make up the managed object class definition.

The Managed Object Class template references the following other templates either directly or indirectly:

- Package
- Parameter
- Attribute
- Attribute Group
- Behaviour
- Action
- Notification

C.2.1 Managed Object Class Template Format

The format of the Managed Object Class template is as follows:

```
<class-label> MANAGED OBJECT CLASS
[DERIVED FROM      <class-label>      [,<class-label>]*;
]

[CHARACTERIZED BY   <package-label>    [,<package-label>]*;
]

[CONDITIONAL PACKAGES <package-label> PRESENT IF condition-definition
                      [,<package-label> PRESENT IF condition-definition]*;
]

REGISTERED AS object-identifier;
```


C.2.2 Managed Object Class Template Constructs

The meaning of each construct in the Managed Object Class template is defined in TABLE C-1.

TABLE C-1 Managed Object Class Template Constructs

Construct	Description
<code><class-label> MANAGED OBJECT CLASS</code>	The <code>MANAGED OBJECT CLASS</code> construct identifies the template as a definition for a managed object class. <code><class-label></code> specifies the name of the managed object class. The <code>REGISTERED AS</code> construct in the template associates a unique object identifier with this label.
<code>DERIVED FROM <class-label> [,<class-label>]*</code>	The <code>DERIVED FROM</code> construct identifies other managed object classes that are base classes of the managed object class that is being defined. Each <code><class-label></code> specifies a base class of the managed object class that is being defined.
<code>CHARACTERIZED BY <package-label> [,<package-label>]*</code>	The <code>CHARACTERIZED BY</code> construct identifies the packages that the managed object class supports. A package defines a set of behaviors, attributes, operations, and notifications. For detailed information on the contents of a package, see Section C.4 “Package Template” on page C-7. Each <code><package-label></code> is the name of a package that the managed object class supports.
<code>CONDITIONAL PACKAGES <package-label> PRESENT IF condition-definition [,<package-label> PRESENT IF condition- definition]</code>	The <code>CONDITIONAL PACKAGES</code> construct identifies any packages that are conditionally supported by the managed object class. Each conditional package will be included in a managed object instance if the condition specified by the <code>PRESENT IF</code> clause for the package is true. Each <code><package-label></code> is the name of a conditional package. Each <code>condition-definition</code> specifies a condition that must be true for the package to be included in a managed object instance when the instance is created.
<code>REGISTERED AS object-identifier</code>	The <code>REGISTERED AS</code> construct identifies the globally unique identifier assigned to the managed object class. <code>object-identifier</code> is replaced with the object identifier (OID) that globally and uniquely identifies the managed object class.

C.3 Name Binding Template

The Name Binding template specifies instantiation and legal parameters for managed objects. Containment, creation, and deletion constraints are initiated from this template.

C.3.1 Name Binding Template Format

The format of the Name Binding template is as follows:

```
<name-binding-label> NAME BINDING
    SUBORDINATE OBJECT CLASS          <class-label> [AND SUBCLASSES];
    NAMED BY SUPERIOR OBJECT CLASS    <class-label> [AND SUBCLASSES];
    WITH ATTRIBUTE                     <attribute-label>;

    [ BEHAVIOUR                        <behaviour-definition-label>
    ]                                  [, <behaviour-definition-label>]*;

    [ CREATE                           [create-modifier [ ,create-modifier]]
    ]                                  [<parameter-label>]*;

    [ DELETE                           [delete-modifier] [<parameter-label>]*;
    ]

REGISTERED AS      object-identifier;
```

C.3.2 Name Binding Template Constructs

The meaning of each construct in the Name Binding template is explained in TABLE C-2.

TABLE C-2 Name Binding Template Constructs

Construct	Description
<code><name-binding-label></code> <code>NAME BINDING</code>	The <code>NAME BINDING</code> construct identifies the template as a definition for a name binding. <code><name-binding-label></code> specifies the name of the name binding. The <code>REGISTERED AS</code> construct in the template associates a unique OID with this label.
<code>SUBORDINATE OBJECT CLASS</code> <code><class-label></code> <code>[AND SUBCLASSES]</code>	The <code>SUBORDINATE OBJECT CLASS</code> construct identifies a managed object class whose instances can be named by using an instance of the managed object class specified in the <code>NAMED BY SUPERIOR OBJECT CLASS</code> construct. <code><class-label></code> identifies the subordinate managed object class. <code>AND SUBCLASSES</code> , if included, specifies that instances of a derived class of the subordinate class can also be named by using an instance of the class specified in the <code>NAMED BY SUPERIOR OBJECT CLASS</code> construct.
<code>NAMED BY SUPERIOR OBJECT CLASS <class-label></code> <code>[AND SUBCLASSES]</code>	The <code>NAMED BY SUPERIOR OBJECT CLASS</code> construct identifies a managed object class whose instances can be used to name an instance of the managed object class specified in the <code>SUBORDINATE OBJECT CLASS</code> construct. <code><class-label></code> identifies the superior managed object class. <code>AND SUBCLASSES</code> , if included, specifies that instances of a derived class of the superior class can also be used to name an instance of the class specified in the <code>SUBORDINATE OBJECT CLASS</code> construct.
<code>WITH ATTRIBUTE</code> <code><attribute-label></code>	The <code>WITH ATTRIBUTE</code> construct identifies the attribute that will be used to form the relative distinguished name (RDN) of an instance of the managed object class specified in the <code>SUBORDINATE OBJECT CLASS</code> construct. <code><attribute-label></code> identifies the attribute used to form the RDN for an instance of the object class specified by the <code>SUBORDINATE OBJECT CLASS</code> construct. For more information on RDNs, refer to Section 2.2.6.2 “Names of Managed Object Instances” on page 2-18.

TABLE C-2 Name Binding Template Constructs (*Continued*)

Construct	Description
BEHAVIOUR <behaviour-definition-label> [, <behaviour-definition-label>]*	<p>The BEHAVIOUR construct specifies any behavior impact that results specifically due to the use of the name binding.</p> <p>Each <behaviour-definition-label> identifies a behavior as defined in an instance of use of the Behaviour template.</p>
CREATE [create-modifier [, create-modifier]] [<parameter-label>]*	<p>The CREATE construct specifies that an instance of managed object class specified by the SUBORDINATE OBJECT CLASS construct can be created by a management operation (normally a CMIS M-CREATE operation).</p> <p>create-modifier specifies permitted options for an M-CREATE operation. The options are as follows:</p> <ul style="list-style-type: none"> • WITH-REFERENCE-OBJECT - Specifies that a reference object may be specified in an M-CREATE operation • WITH-AUTOMATIC-INSTANCE-NAMING - Specifies that the object instance name can be omitted from the M-CREATE operation <p><parameter-label> identifies name binding error parameters associated with create operations.</p>
DELETE [delete-modifier] [<parameter-label>]*	<p>The DELETE construct specifies that an instance of managed object class specified by the SUBORDINATE OBJECT CLASS construct can be deleted by a management operation (normally a CMIS M-DELETE operation).</p> <p>delete-modifier specifies behavior for a managed object instance when the instance is deleted. The options are as follows:</p> <ul style="list-style-type: none"> • ONLY-IF-NO-CONTAINED-OBJECTS - Specifies that a delete operation will fail and return an error if there are any contained managed object instances under the instance being deleted • DELETES-CONTAINED-OBJECTS - Specifies that a delete operation will fail and return an error if any of the managed object instances directly or indirectly under the instance being deleted are subject to the ONLY-IF-NO-CONTAINED-OBJECTS delete modifier <p><parameter-label> identifies name binding error parameters associated with delete operations.</p>
REGISTERED AS object-identifier	<p>The REGISTERED AS construct identifies the globally unique identifier assigned to the name binding. object-identifier is replaced with the OID that globally and uniquely identifies the name binding.</p>

C.4 Package Template

The Package template is defines a combination of behavior definitions, attributes, attribute groups, operations, actions, notifications, and parameters for later inclusion in a managed object class template. A package can be referenced by more than one managed object class definition.

C.4.1 Package Template Format

The format of the Package template is as follows:

```
<package-label> PACKAGE
    [BEHAVIOUR          <behaviour-definition-label>
                        [, <behaviour-definition-label>]*;
    ]

    [ATTRIBUTES         <attribute-label> propertyList [<parameter-label>] *
                        [, <attribute-label> propertyList [<parameter-label>]*]*;
    ]

    [ATTRIBUTE GROUPS  <group-label> [<attribute-label>]*
                        [, <group-label> [<attribute-label>]*]*;
    ]

    [ACTIONS           <action-label> [<parameter-label>] *
                        [, <action-label> [<parameter-label>]*]*;
    ]

    [NOTIFICATIONS     <notification-label> [<parameter-label>]*
                        [, <notification-label> [<parameter-label>]*]*;
    ]

    [REGISTERED AS object-identifier];
```

C.4.2 Package Template Constructs

The meaning of each construct in the Package template is explained in TABLE C-3.

TABLE C-3 Package Template Constructs

Construct	Description
<code><package-label></code> PACKAGE	The PACKAGE construct identifies the template as a definition for a package. <code><package-label></code> specifies the name of the package. The REGISTERED AS construct in the template associates a unique OID with this label.
BEHAVIOUR <code><behaviour-definition-label></code> [<code><behaviour-definition-label></code>]*	The BEHAVIOUR construct enables the semantics of the package to be completely described. This construct relates the external view of aspects of a managed object, such as its operations and notifications, to its internal operation. Each <code><behaviour-definition-label></code> identifies a behavior as defined in an instance of use of the Behaviour template.
ATTRIBUTES <code><attribute-label></code> propertyList [<code><parameter-label></code>] *[<code><attribute-label></code> propertyList [<code><parameter-label></code>]*]*	The ATTRIBUTES construct defines operations permitted on a managed object with respect to its attributes. Each <code><attribute-label></code> specifies the name of an attribute. Each propertyList specifies the operations permitted on a managed object with respect to the attribute associated with the propertyList. The format of a propertyList is defined in Section C.4.3 “PropertyList Supporting Production” on page C-9. Each <code><parameter-label></code> specifies an error parameter specific to the managed object class. This parameter is associated with management operations on the attribute.
ATTRIBUTE GROUPS <code><group-label></code> [<code><attribute-label></code>]* [<code><group-label></code> [<code><attribute-label></code>]*]*	The ATTRIBUTE GROUPS construct enables a set of attribute groups to be identified as part of the package. Each <code><group-label></code> specifies the name of an attribute group. Each <code><attribute-label></code> specifies the name of an attribute in an attribute group.

TABLE C-3 Package Template Constructs *(Continued)*

Construct	Description
ACTIONS <action-label> [<parameter-label>] * [,<action-label> [<parameter-label>]*]*	The ACTIONS construct specifies a set of action definitions that are included in the package. Each <action-label> specifies the name of an action. Each <parameter-label> specifies the name of an action information, action reply or error parameter.
NOTIFICATIONS <notification-label> [<parameter-label>]* [,<notification-label [<parameter-label>]*]*	The NOTIFICATIONS construct specifies the notifications that are included in the package. Each <notification-label> specifies the name of a notification. Each <parameter-label> specifies the name of a parameter.
REGISTERED AS object-identifier	The REGISTERED AS construct identifies the globally unique identifier assigned to the package. object-identifier is replaced with the OID that globally and uniquely identifies the package.

C.4.3 PropertyList Supporting Production

The format of the propertyList supporting production is as follows:

```
[REPLACE-WITH-DEFAULT]
[DEFAULT VALUE      value-specifier]
[INITIAL VALUE      value-specifier]
[PERMITTED VALUES  type-reference]
[REQUIRED VALUES  type-reference]
[get-replace]
[add-remove]
[SET-BY-CREATE]
[NO-MODIFY]
```

The items in the `propertyList` supporting production are defined in TABLE C-4.

TABLE C-4 `propertyList` Supporting Production Definitions

Item	Description
<code>REPLACE-WITH-DEFAULT</code>	Specifies that a management operation can set the attribute to a default value. If this property is specified but the <code>DEFAULT VALUE</code> property is not specified, the default value is determined by other means local to the management system.
<code>DEFAULT VALUE</code> <code>value-specifier</code>	Specifies that the attribute has a default value given by <code>value-specifier</code> .
<code>INITIAL VALUE</code> <code>value-specifier</code>	Specifies that the attribute must be set at creation time to the value given by <code>value-specifier</code> .
<code>PERMITTED VALUES</code> <code>type-reference</code>	Specifies that permitted values of the attribute are restricted to those given by <code>type-reference</code> .
<code>REQUIRED VALUES</code> <code>type-reference</code>	Specifies that the attribute shall be able to take any values given by <code>type-reference</code> . This property defines the value set required for conformance.
<code>value-specifier</code>	Specifies the value in a <code>DEFAULT VALUE</code> or <code>INITIAL VALUE</code> property. <code>value-specifier</code> is one of the following: <ul style="list-style-type: none"> <code>value-reference</code> - The value is a fully qualified reference to a data value defined in an ASN.1 module, that is: <code>ASN.1-Module-Name.data-value-label</code>. <code>DERIVATION RULE <behaviour-definition-label></code> - The value is determined by the derivation rule specified in <code><behaviour-definition-label></code>. <code><behaviour-definition-label></code> identifies a behavior as defined in an instance of use of the Behaviour template.
<code>type-reference</code>	Specifies the values in a <code>PERMITTED VALUES</code> or a <code>REQUIRED VALUES</code> property. <code>type-reference</code> is a fully qualified reference to a data type defined in an ASN.1 module, that is: <code>ASN.1-Module-Name.data-type-label</code> .
<code>get-replace</code>	Specifies the management operations allowed on the attribute. <code>get-replace</code> is one of the following: <ul style="list-style-type: none"> <code>GET</code> - A management operation can read the attribute. <code>REPLACE</code> - A management operation can set the attribute. <code>GET-REPLACE</code> - A management operation can read or set the attribute.

TABLE C-4 `propertyList` Supporting Production Definitions (*Continued*)

Item	Description
<code>add-remove</code>	Specifies, for a multi-valued attribute only, how a management operation is allowed to add or remove values in the attribute. <code>add-remove</code> is one of the following: <ul style="list-style-type: none">• <code>ADD</code> - A management operation can add a value to the attribute.• <code>REMOVE</code> - A management operation can remove a value from the attribute.• <code>ADD-REMOVE</code> - A management operation can add a value to or remove a value from the attribute.
<code>SET-BY-CREATE</code>	Specifies that the attribute may be set by a creation operation (normally a CMIS <code>M-CREATE</code> operation). The <code>SET-BY-CREATE</code> property is meaningful only if the name binding of the managed object instances supports the creation operation.
<code>NO-MODIFY</code>	Specifies that the attribute cannot be modified in the class that has this property and in all subclasses and compatible managed objects. This property is not allowed in a managed object class definition that has the <code>REPLACE</code> , <code>GET-REPLACE</code> , <code>ADD</code> , <code>REMOVE</code> , or <code>ADD-REMOVE</code> properties on the same attribute.

C.5 Attribute Template

The Attribute template defines attributes used by GDMO classes. A single attribute can be referenced by more than one managed object class definition.

The Attribute template defines individual attribute types. If desired these attribute types can be combined in a group by using the Attribute Group template.

C.5.1 Attribute Template Format

The format of the Attribute template is as follows:

```
<attribute-label> ATTRIBUTE
    derived-or-with-syntax-choice;

    [MATCHES FOR qualifier [,qualifier]*;]

    [BEHAVIOUR    <behaviour-definition-label> [,<behaviour-definition-label>]*;
    ]

    [PARAMETER    <parameter-label> [<parameter-label>]*;
    ]

[REGISTERED AS object-identifier];
```

C.5.2 Attribute Template Constructs

The meaning of each construct in the Attribute template is explained in TABLE C-5.

TABLE C-5 Attribute Template Constructs

Construct	Description
<attribute-label> ATTRIBUTE	The ATTRIBUTE construct identifies the template as a definition for an attribute. <attribute-label> specifies the name of the attribute. The REGISTERED AS construct in the template associates a unique object identifier with this label.
derived-or-with-syntax-choice	This construct specifies how the attribute is defined. derived-or-with-syntax-choice is one of the following mutually exclusive options: <ul style="list-style-type: none">• DERIVED FROM <attribute-label> - The attribute is based on the definition referenced by <attribute-label>.• WITH ATTRIBUTE SYNTAX <type-reference> - The data type of the attribute is the ASN.1 type specified by <type-reference>.

TABLE C-5 Attribute Template Constructs (*Continued*)

Construct	Description
<code>MATCHES FOR</code> <i>qualifier</i> [, <i>qualifier</i>]*	The <code>MATCHES FOR</code> construct defines the types of tests that may be applied to the value of the attribute by using a filter operation. Each <i>qualifier</i> is one of the following: <ul style="list-style-type: none"> • <code>EQUALITY</code> - The attribute value can be tested for equality against a given value. • <code>ORDERING</code> - The attribute value can be tested against a given value to determine which value is greater. • <code>SUBSTRINGS</code> - The attribute value can be tested against a given substring to determine if the attribute value is present in the substring. • <code>SET-COMPARISON</code> - The attribute value can be tested against a given value to determine if the attribute value is a subset or superset of the given value. • <code>SET-INTERSECTION</code> - The attribute value can be tested against a given value to determine if there is a nonnull set intersection between the two values.
<code>BEHAVIOUR</code> <behaviour-definition-label> [,<behaviour-definition-label>]*	The <code>BEHAVIOUR</code> construct specifies behavior that is generic to this attribute. Each <behaviour-definition-label> specifies an instance of use of the Behaviour template.
<code>PARAMETER</code> <parameter-label> [<parameter-label>]*	The <code>PARAMETER</code> construct associates parameters with the behavior of the attribute for the definition of processing failures. Each <parameter-label> specifies a processing failure parameter for the attribute.
<code>REGISTERED AS</code> object-identifier	The <code>REGISTERED AS</code> construct identifies the globally unique identifier assigned to the attribute. object-identifier is replaced with the OID that globally and uniquely identifies the attribute.

C.6 Action Template

The Action template defines actions that can be performed by a managed object class. The Action template defines actions that are performed by using the CMIS `M-ACTION` service. The template does not include actions or behavior defined for other CMIS services, such as `M-GET`, or `M-SET`, for example. An action can be referenced by more than one managed object class.

The Action template defines the behavior and syntax of an action. The syntax definitions specify the contents of the action information and action reply fields in CMIS action requests and responses.

C.6.1 Action Template Format

The format of the Action template is as follows:

```
<action-label> ACTION
    [BEHAVIOUR                <behaviour-definition-label>
                                [, <behaviour-definition-label>]*;

    ]

    [MODE CONFIRMED;
    ]

    [PARAMETERS                <parameter-label> [<parameter-label>]*;
    ]

    [WITH INFORMATION SYNTAX    type-reference;
    ]

    [WITH REPLY SYNTAX          type-reference;]
REGISTERED AS object-identifier;
```

C.6.2 Action Template Constructs

The meaning of each construct the Action template is explained in TABLE C-6.

TABLE C-6 Action Template Constructs

Construct	Description
<action-label> ACTION	The ACTION construct identifies the template as a definition for an attribute. <action-label> specifies the name of the action. The REGISTERED AS construct in the template associates a unique object identifier with this label.
BEHAVIOUR <behaviour-definition-label> [, <behaviour-definition-label>]*	The BEHAVIOUR construct defines the behavior of the action, the parameters that shall be passed to the action, the results the action may generate, and the meaning of the results. Each <behaviour-definition-label> specifies an instance of use of the Behaviour template.

TABLE C-6 Action Template Constructs (*Continued*)

Construct	Description
MODE CONFIRMED	The MODE CONFIRMED construct specifies that an action is confirmed. If this construct is not specified, the choice of whether the action is confirmed is left to the entity acting in the manager role.
PARAMETERS <parameter-label> [<parameter-label>]*	The PARAMETERS construct specifies action information, action reply parameters, or processing information for the action. Each <parameter-label> specifies the name of a parameter.
WITH INFORMATION SYNTAX type-reference	The WITH INFORMATION SYNTAX construct specifies the action information associated with this type of action. type-reference defines the data type of the action information. It is a fully qualified reference to a data type defined in an ASN.1 module, that is: ASN.1-Module-Name.data-type-label.
WITH REPLY SYNTAX type-reference	The WITH REPLY SYNTAX construct specifies the action reply information associated with this type of action. type-reference defines the data type of the action reply information. It is a fully qualified reference to a data type defined in an ASN.1 module, that is: ASN.1-Module-Name.data-type-label.
REGISTERED AS object-identifier	The REGISTERED AS construct identifies the globally unique identifier assigned to the action. object-identifier is replaced with the OID that globally and uniquely identifies the action.

C.7 Notification Template

The Notification template defines notifications that can be emitted by a managed object class. A notification can be referenced by more than one managed object class.

The Notification template defines the behavior and syntax of a notification. The syntax definitions specify the contents of the event information and event reply fields in CMIP event report requests and responses.

C.7.1 Notification Template Format

The format of the Notification template is as follows:

```
<notification-label> NOTIFICATION
    [BEHAVIOUR                <behaviour-definition-label>
                                [, <behaviour-definition-label>]*;
    ]

    [PARAMETERS                <parameter-label> [<parameter-label>]*;
    ]

    [WITH INFORMATION SYNTAX   type-reference
      [AND ATTRIBUTE IDS       <field-name> <attribute-label>
                                [, <field-name> <attribute-label>]*
      ] ;
    ]

    [WITH REPLY SYNTAX         type-reference;
    ]

REGISTERED AS object-identifier;
```

C.7.2 Notification Template Constructs

The meaning of each construct in the Notification template is explained in TABLE C-7.

TABLE C-7 Notification Template Constructs

Construct	Description
NOTIFICATION <notification-label>	The NOTIFICATION construct identifies the template as a definition for a notification. <notification-label> specifies the name of the notification. The REGISTERED AS construct in the template associates a unique object identifier with this label.
BEHAVIOUR <behaviour-definition-label> [, <behaviour-definition-label>]*	The BEHAVIOUR construct defines the behavior of the notification, the data that shall be specified with the notification, the results the notification may generate, and the meaning of the results. Each <behaviour-definition-label> specifies an instance of use of the Behaviour template.

TABLE C-7 Notification Template Constructs (*Continued*)

Construct	Description
PARAMETERS <code><parameter-label></code> <code>[<parameter-label>]*</code>	<p>The PARAMETERS construct specifies event information, event reply parameters, or processing information for the notification.</p> <p>Each <code><parameter-label></code> specifies the name of a parameter.</p>
WITH INFORMATION SYNTAX <code>type-reference</code> <code>[AND ATTRIBUTE IDS</code> <code><field-name></code> <code><attribute-label></code> <code>[,<field-name></code> <code><attribute-label>]*]</code>	<p>The WITH INFORMATION SYNTAX construct specifies the event information associated with this type of notification. <code>type-reference</code> defines the data type of the action information. It is a fully qualified reference to a data type defined in an ASN.1 module, that is: <code>ASN.1-Module-Name.data-type-label</code>.</p> <p>The AND ATTRIBUTE IDS option is typically used to specify the attributes contained within a SET or SEQUENCE data type, when a SET or SEQUENCE data type is referenced by the WITH INFORMATION SYNTAX construct. Each <code><field-name></code> specifies a label that is defined within the same ASN.1 module as specified in the WITH INFORMATION SYNTAX construct. The field name label within the ASN.1 module is used to label the data type that specifies the type of the attribute specified by <code><attribute-label></code>.</p> <p>Each <code><attribute-label></code> specifies an attribute that is defined within a GDMO module. The definition of this attribute must reference the data type specified by the <code><field-name></code>.</p>
WITH REPLY SYNTAX <code>type-reference</code>	<p>The WITH REPLY SYNTAX construct specifies the event reply information associated with this type of notification. <code>type-reference</code> defines the data type of the event reply information. It is a fully qualified reference to a data type defined in an ASN.1 module, that is: <code>ASN.1-Module-Name.data-type-label</code>.</p>
REGISTERED AS <code>object-identifier</code>	<p>The REGISTERED AS construct identifies the globally unique identifier assigned to the notification. <code>object-identifier</code> is replaced with the OID that globally and uniquely identifies the notification.</p>

C.8 Parameter Template

The Parameter template defines parameter syntaxes for later inclusion in Package, Attribute, Action, and Notification templates. A parameter can be referenced by one or more of each of these templates.

The Parameter template specifies and registers the parameter syntaxes and associated behavior that may be associated with particular attributes, operations, and notifications within Package, Attribute, Action, and Notification templates. The type specified in a Parameter template is used to fill in the `ANY DEFINED BY oid` construct.

C.8.1 Parameter Template Format

The format of the Parameter template is as follows:

```
<parameter-label> PARAMETER
    CONTEXT                context-type;

    syntax-or-attribute-choice;

    [BEHAVIOUR              <behaviour-definition-label>
      [, <behaviour-definition-label>]*;
    ]

    [REGISTERED AS object-identifier];
```


C.8.2 Parameter Template Constructs

The meaning of each construct in the Parameter template is explained in TABLE C-8.

TABLE C-8 Parameter Template Constructs

Construct	Description
<code><parameter-label></code> PARAMETER	The PARAMETER construct identifies the template as a definition for a parameter. <code><parameter-label></code> specifies the name of the parameter. The REGISTERED AS construct in the template associates a unique object identifier with this label.
CONTEXT context-type	The CONTEXT construct identifies the context in which this parameter is applicable. context-type specifies the context for the parameter definition. context-type can take on one of the following values: <ul style="list-style-type: none">context-keyword - A reference to a context defined externally to the template. context-keyword has the form: type-reference.<identifier>, where type-reference is a fully qualified reference to a data type defined in an ASN.1 module (that is: ASN.1-Module-Name.data-type-label) and <identifier> is the name of one of the fields in that data type.ACTION-INFO - A CMIS action information parameter.ACTION-REPLY - A CMIS action reply parameter.EVENT-INFO - A CMIS event information parameter.EVENT-REPLY - A CMIS event reply parameter.SPECIFIC-ERROR - A CMIS processing failure error.
syntax-or-attribute-choice	This construct specifies how the data type associated with the context of the parameter is defined. syntax-or-attribute-choice is one of the following mutually exclusive options: <ul style="list-style-type: none">ATTRIBUTE <attribute-label> - The syntax and OID of the attribute specified by <attribute-label> are used as the syntax and OID of the parameter.WITH SYNTAX type-reference - The ASN.1 type of the parameter is that specified by type-reference. type-reference is a fully qualified reference to a data type defined in an ASN.1 module, that is: ASN.1-Module-Name.data-type-label.

TABLE C-8 Parameter Template Constructs (*Continued*)

Construct	Description
BEHAVIOUR <behaviour-definition-label> [, <behaviour-definition-label>]*	The BEHAVIOUR construct specifies any behavior or semantics associated with the parameter. If the ATTRIBUTE construct is used, the BEHAVIOUR construct does not modify the behavior of the attribute. Each <behaviour-definition-label> specifies an instance of use of the Behaviour template.
REGISTERED AS object-identifier	The REGISTERED AS construct identifies the globally unique identifier assigned to the parameter. object-identifier is replaced with the OID that globally and uniquely identifies the parameter.

C.9 Attribute Group Template

The Attribute Group template defines one or more attributes that can be referenced as a group. A managed object class definition can include all attributes of a group by referencing the group, rather than referencing each attribute individually. More than one managed object class definition can reference an attribute group.

Attribute groups make it easier to collectively perform operations on a large number of individual attributes.

C.9.1 Attribute Group Template Format

The format of the Attribute Group template is as follows:

```
<group-label> ATTRIBUTE GROUP
  [GROUP ELEMENTS                <attribute-label> [, <attribute-label>]*;
  ]

  [FIXED;
  ]

  [DESCRIPTION                    delimited-string;
  ]

REGISTERED AS object-identifier;
```

C.9.2 Attribute Group Template Constructs

The meaning of each construct in the Attribute Group template is explained in TABLE C-9.

TABLE C-9 Attribute Group Template Constructs

Construct	Description
<code><group-label></code> ATTRIBUTE GROUP	The ATTRIBUTE GROUP construct identifies the template as a definition for an attribute group. <code><group-label></code> specifies the name of the attribute group. The REGISTERED AS construct in the template associates a unique object identifier with this label.
GROUP ELEMENTS <code><attribute-label></code> <code>[,<attribute-label>]*</code>	The GROUP ELEMENTS construct specifies the attributes that are members of this attribute group. Each <code><attribute-label></code> specifies an attribute in the group.
FIXED	The FIXED construct limits the membership of the attribute group to only those attributes specified by the GROUP ELEMENTS construct. If the FIXED construct is omitted, the group is made extensible. The group can be extended by adding attributes that are referenced in Package templates.
DESCRIPTION delimited-string	The DESCRIPTION construct provides a description of the semantics of the attribute group, for example 'Group of all state attributes in the managed object'.
REGISTERED AS object-identifier	The REGISTERED AS construct identifies the globally unique identifier assigned to the attribute group. <code>object-identifier</code> is replaced with the OID that globally and uniquely identifies the attribute group.

C.10 Behaviour Template

The Behaviour template describes the expected behavior of:

- Managed object classes
- Name bindings
- Parameters
- Attributes
- Actions
- Notifications

The behavior may be defined by readable text, high level languages, formal description techniques, references to standard constructs, or by any combination of the preceding definition methods.

C.10.1 Behaviour Template Format

The format of the Behaviour template is as follows:

```
<behaviour-definition-label> BEHAVIOUR
    [DEFINED AS                delimited-string;
    ]
```

C.10.2 Behaviour Template Constructs

The meaning of each construct in the Behaviour template is explained in TABLE C-10.

TABLE C-10 Behaviour Template Constructs

Construct	Description
BEHAVIOUR <behaviour-definition-label>	The BEHAVIOUR construct identifies the template as a definition for a behavior. <behaviour-definition-label> specifies the name of the behavior.
DEFINED AS delimited-string	The DEFINED AS construct defines the behavior. delimited-string contains the text of the behavior.

Index

SYMBOLS

! operator, 4-2

A

abortAssociation enforcement action, 12-30

Abstract Syntax Notation #1 (ASN.1)

- introduction, 2-23

- basic encoding rules (BER), 15-8 to 15-9

- em_debug message types, 15-15

- syntax and logic, verifying, 15-2

abstracting managed object classes, 2-7 to 2-8

ACAccessControlRules class

- get_access_control_switch function, 12-9 to 12-10

- get_default_access function, 12-33

- get_default_event_access function, 12-33

- get_denial_granularity function, 12-35

- get_domain_identity function, 12-36

- get_trusted_host_list function, 12-34

- set_access_control_switch function, 12-9 to 12-10

ACAccessUserList class, 12-13

ACAppFeatureContainer class, 12-17

ACApplication class, 12-17

ACApplicationContainer class, 12-16

ACApplicationFeature class, 12-17

ACApplicationFeatureList class, 12-17

access control

- overview, 1-10 to 1-12

- activating, 12-9 to 12-10

- application level

 - introduction, 12-2

 - enforcing predefined rules, 12-6, 12-7 to 12-8
 - privilege groups, adding applications to, 12-18

- application-feature level

 - introduction, 12-2

 - enforcing predefined rules, 12-7 to 12-8

 - getting feature list, 12-16 to 12-18

 - privilege groups, adding features to, 12-18

- deactivating, 12-9 to 12-10

- decision and enforcement functions, 12-44

- defaults, getting, 12-32 to 12-36

- denial granularity, 12-35

- domains

 - generally, 12-36

 - specifying for secure MPAs, 12-42

- em_debug message types, 15-15

- enforcement actions

 - defining, 12-29 to 12-30

 - events, 12-33 to 12-34

 - management operations, 12-33

- error handling

 - generally, 12-31 to 12-32

 - rule creation, 12-27

 - target creation, 12-20

- events

 - introduction, 12-3

 - auxiliary object, 12-37, 12-38

 - enforcement actions, 12-33 to 12-34

 - log owners, assigning, 12-36 to 12-39

 - log server, enabling, 12-39

- examples, A-9 to A-10

- managed-object level

 - introduction, 12-3

 - denial, handling, 12-8

 - granularity, of denial, 12-35

- operations permitted, defining, 12-21 to 12-22
- rules, 12-29, 12-33
- selecting objects for, 12-18 to 12-21, 12-23 to 12-25
- MPAs
 - introduction, 12-3
 - ACE class, 12-42 to 12-43
 - AuxServerUtils class, 12-44
 - decision and enforcement functions, 12-44
 - domains, 12-42
 - processing information in events, 12-43 to 12-44
 - services required, 12-42 to 12-43
 - updating access control information, 12-40 to 12-41
- privilege groups
 - introduction, 12-10
 - applications, adding, 12-18
 - creating, 12-11 to 12-12
 - features, adding, 12-18
 - MIS, adding to, 12-11
 - predefined, 12-10
 - rules, adding to, 12-28 to 12-29
 - storing persistently, 12-14 to 12-15
 - users, adding, 12-14 to 12-15
- rules
 - introduction, 12-26
 - creating, 12-26 to 12-28
 - defining, 12-4 to 12-6
 - enforcing, 12-4, 12-6 to 12-8
 - MIS, adding to, 12-27
 - predefined, 12-26
 - privilege groups, adding, 12-28 to 12-29
 - storing persistently, 12-30 to 12-31
 - targets, adding, 12-29
- targets
 - introduction, 12-18 to 12-19
 - creating, 12-19 to 12-21
 - operations permitted, defining, 12-21 to 12-22
 - rules, adding to, 12-29
- trusted hosts, 12-34
- users
 - introduction, 12-10
 - creating, 12-12
 - MIS, adding to, 12-13 to 12-14
 - privilege groups, adding to, 12-14 to 12-15
- access control API
 - introduction, 1-16
 - error-handling functions, 12-31 to 12-32
 - examples, A-9
- access control engine API

- introduction, 1-16
- examples, A-9
- access_type property, MorfBuilder class, 9-37 to 9-38
- ACDbObject class
 - constructor, 12-37
 - create function, 12-37
 - exists function, 12-37
 - set_auxobject_owner function, 12-38
 - store function, 12-38
- ACE class, 12-42 to 12-43
- ACFilter defined type, 12-24
- ACGroup class
 - add_application function, 12-18
 - add_application_feature function, 12-18
 - add_group_member function, 12-14
 - constructor, 12-11
 - create function, 12-11
 - exists function, 12-11
 - store function, 12-15
- ACInterface class
 - get_access_user_list function, 12-13
 - get_application_container function, 12-16
- ACMOCList defined type, 12-25
- ACMOIList defined type, 12-23
- ACOperationsList defined type, 12-21
- ACRule class
 - add_group function, 12-28
 - add_targets function, 12-29
 - constructor, 12-27
 - create function, 12-27
 - exists function, 12-27
 - get_error_string function, 12-27
 - get_error_type function, 12-27
 - set_enforcement_action function, 12-30
 - store function, 12-30
- ACScope class, 12-24
- ACTargets class
 - introduction, 12-19
 - add_moc function, 12-24
 - add_moi function, 12-23
 - constructor, 12-19 to 12-20
 - create function, 12-20
 - exists function, 12-20
 - get_error_string function, 12-20
 - get_error_type function, 12-20
 - set_filter function, 12-24
 - set_moc_list function, 12-25
 - set_moi_list function, 12-23

- set_operations_list function, 12-22
 - set_scope function, 12-24
 - store function, 12-25
- Action GDMO template, C-13 to C-15
- action operation, targets, 12-22
- actions
 - debugging, 15-2
 - defining, 2-11
 - em_debug message types, 15-15
 - on managed objects
 - generally, 5-23 to 5-25
 - asynchronous, 8-3
 - metadata retrieval, 5-29 to 5-30
 - on object collections
 - generally, 6-20
 - asynchronous, 8-4
 - CMIS, 8-10
 - replies to, information contained in, 8-17
 - retrieving from MDR, 5-30
 - timeouts, 6-20
- Actions menu, Network Views tool, customizing, 16-6 to 16-8
- activating
 - access control, 12-9 to 12-10
 - Image instances
 - generally, 5-4
 - asynchronously, 8-3, 8-4
 - in object collections, 6-16 to 6-17, 8-4
- activation, topology types, customizing, 16-9 to 16-10
- ACUser class, 12-12
- ADD operation
 - asynchronous CMIS M-SET request, 8-9
 - Image class and, 5-21
- add_application function, ACGroup class, 12-18
- add_application_feature function, ACGroup class, 12-18
- add_group function, ACRule class, 12-28
- add_group_member function, ACGroup class, 12-14
- add_moc function, ACTargets class, 12-24
- add_moi function, ACTargets class, 12-23
- add_targets function, ACRule class, 12-29
- add_user function, ACAccessUserList class, 12-13
- addMember operation, targets, 12-22
- Administration window, customizing, 16-2 to 16-3
- AdministrativeState, 10-13
- agent
 - directly access managed resource, 17-2
 - indirectly access managed resource, 17-2
 - manager-agent model, 17-1
 - process that accesses the managed object, 17-1
 - process that accesses the managed resource, 17-1
 - process that collects data, 17-1
- agent role, Solstice EM in, 2-37 to 2-38
- agents
 - introduction, 2-3
 - filters, support for, 15-20 to 15-23
 - hierarchical arrangement, 2-4
 - in ISO model, 1-2
 - scopes, support for, 15-20 to 15-23
 - simulating, 5-30 to 5-35
- alarm log manager, em_debug message types, 15-16
- alarm services, em_debug message types, 15-15
- Alarms tool, customizing, 16-4 to 16-5
- Album class
 - introduction, 6-2
 - all_boot function, 8-4
 - all_call function, 6-20
 - all_create function, 6-18
 - all_create_within function, 8-4
 - all_destroy function, 6-18
 - all_set_dbl function, 6-19
 - all_set_gint function, 6-19
 - all_set_long function, 6-19
 - all_set_raw function, 6-19
 - all_set_str function, 6-19
 - all_shutdown function, 8-4
 - all_start functions, 8-4
 - all_store function, 6-20
 - asynchronous functions, 8-4
 - AUTOIMAGE property, 6-16
 - CMIS operations supported, 8-5
 - constructor, 6-2
 - derive function
 - generally, 6-5
 - restrictions, 6-15
 - exclude function, 6-15
 - first_image function, 6-22
 - include function, 6-15
 - instance associated with event, getting, 7-7
 - nicknames, 6-2
 - performance considerations, 13-3
 - set_derivation function, 6-3
 - set_prop function, 6-14, 6-17, 6-21
 - start_derive function, 8-4
 - start_m_action function, 8-10
 - start_m_action_raw function, 8-10
 - start_m_get function, 8-7
 - start_m_set function, 8-7 to 8-10

- TRACKMODE property, 6-14
 - when function, 7-4 to 7-5
- AlbumImage class
 - introduction, 6-21
 - next_album function, 6-23
 - next_image function, 6-22
- all_boot function, Album class, 8-4
- all_call function, Album class, 6-20
- all_create function, Album class, 6-18
- all_create_within function, Album class, 8-4
- all_destroy function, Album class, 6-18
- ALL_LEVELS scope value, 12-24
- ALL_LEVELS_EXCEPT_BASE scope value, 12-24
- all_set_dbl function, Album class, 6-19
- all_set_gint function, Album class, 6-19
- all_set_long function, Album class, 6-19
- all_set_raw function, Album class, 6-19
- all_set_str function, Album class, 6-19
- all_shutdown function, Album class, 8-4
- all_start functions, Album class, 8-4
- all_store function, Album class, 6-20
- allocating OIDs, guidelines for, 2-32 to 2-33
- allow enforcement action, 12-30
- and keyword, filters, 6-9
- annotation secretary, em_debug message types, 15-15
- ANY ASN.1 type
 - definition, 2-26
 - creating Morf instances for, 9-7 to 9-8
 - formatting string representation of, 9-27
 - string representation, Morf class, 9-25
- ANY DEFINED BY ASN.1 type
 - creating Morf instances for, 9-7 to 9-8
 - formatting string representation of, 9-27
- anyString keyword, filters, 6-11
- APIs (application programming interfaces) of Solstice EM
 - architecture, 1-14
 - uses of, 1-1
- application class ASN.1 tags, 15-9
- application context, 7-13
- application launcher, *See* tools windows
- application programming interfaces (APIs) of Solstice EM
 - architecture, 1-14
 - uses of, 1-1
- APPLICATION_OBJNAME property, 7-23
- applications
 - access control
 - introduction, 12-2
 - enforcing predefined rules, 12-6, 12-7 to 12-8
 - privilege groups, adding applications to, 12-18
- access to managed objects
 - generally, 3-1 to 3-3
 - asynchronous, 8-3
- compiling, guidelines for, 14-1 to 14-8
- data component, 1-3
- development process
 - overview, 1-4
 - debugging, 1-22, 15-1 to 15-26
 - examples, A-2 to A-3, A-8
 - high-level design, 1-4 to 1-13
 - implementation, 1-19 to 1-22
 - integration, 1-22
 - low-level design, 1-13 to 1-18
 - requirements analysis, 1-4 to 1-13
 - system testing, 1-22
 - unit testing, 1-22, 15-1 to 15-26
- direct access to databases, 3-5 to 3-10
- error handling, 4-1 to 4-3
- feature-level access control
 - introduction, 12-2
 - enforcing predefined rules, 12-7 to 12-8
 - getting feature list, 12-16 to 12-18
 - privilege groups, adding applications to, 12-18
- GUI component, 1-3
- integrating, 16-1 to 16-10
- linking, guidelines for, 14-1 to 14-8
- performance, enhancing, 13-1 to 13-6
- programming model, 1-2 to 1-3
- Solstice EM API component, 1-2 to 1-3
- starting
 - overview, 1-8 to 1-9
 - from Actions menu, Network Views tool, 16-6 to 16-8
 - by double clicking topology nodes, 16-9 to 16-10
 - from Tools menu, Solstice EM tools, 16-4 to 16-5
 - from tools windows, 16-1 to 16-3
- tuning, 13-1 to 13-2
- types developed with Solstice EM APIs, 1-1
- user interaction with, 1-7 to 1-9
- application-to-application API
 - introduction, 1-16
 - examples, A-12
- architecture, of Solstice EM APIs, 1-14
- arguments
 - Actions menu applications, 16-7
 - Tools menu applications, 16-5
 - tools window applications, 16-3

- topology type activation, 16-9
- ASN.1
 - printing values, 10-28
 - sanity check, 10-6
- ASN.1 (Abstract Syntax Notation #1)
 - introduction, 2-23
 - basic encoding rules (BER), 15-8 to 15-9
 - em_debug message types, 15-15
 - syntax and logic, verifying, 15-2
- ASN.1 modules
 - examples, A-11
 - exporting, 2-29
 - format of, 2-23 to 2-24
 - importing, 2-28 to 2-29
 - information in, retrieving from MDR, 5-29
 - names, retrieving from MDR, 5-29
- ASN.1 tags, 15-9 to 15-10
- ASN.1 types
 - allowed values
 - defining, 2-27
 - getting, 9-18 to 9-21
 - complex
 - checking value last set, 5-22
 - getting, 5-17
 - setting, 5-19, 6-19
 - definitions, format of, 2-24 to 2-26
 - of Morf instance, getting, 9-16
 - tag numbers, 15-10
 - universal types, 2-26
- ASN.1 values
 - decoding
 - introduction, 9-1
 - ENUMERATED values, 9-29
 - examples, A-8
 - lists, 9-27 to 9-28
 - OBJECT IDENTIFIER values, 9-29
 - scalars, 9-28 to 9-31
 - SEQUENCE values, 9-27 to 9-28
 - SET values, 9-27 to 9-28
 - as strings, 9-25 to 9-27
 - defining in ASN.1 module specification, 2-27 to 2-28
 - encoding
 - introduction, 9-1
 - ANY and ANY DEFINED BY values, 9-7 to 9-8
 - CHOICE values, 9-6, 9-35 to 9-37
 - examples, A-8
 - lists, 9-2 to 9-3
 - MorfBuilder class, by using, 9-32 to 9-39
 - scalars, 9-5
 - SEQUENCE and SEQUENCE OF values, 9-8 to 9-9, 9-37 to 9-38
 - SET and SET OF values, 9-8 to 9-9
 - from string data, 9-2 to 9-4
 - extracting
 - from lists, 9-27 to 9-29
 - from scalars, 9-28 to 9-31
 - formatting, 9-26 to 9-27
 - parsing
 - introduction, 9-9 to 9-10
 - BIT STRING values, 9-16, 9-17 to 9-18, 9-21 to 9-23
 - CHOICE values, 9-12 to 9-13
 - ENUMERATED values, 9-16 to 9-18
 - example, 9-23 to 9-24
 - lists, 9-13 to 9-15
 - OCTET STRING values, 9-21 to 9-23
 - REAL values, 9-18 to 9-21
 - SEQUENCE OF values, 9-21 to 9-23
 - SET OF values, 9-21 to 9-23
 - type, getting, 9-16
 - ranges allowed for a type
 - defining, 2-27
 - getting, 9-18 to 9-21
- Asn1ParsedValue class, 9-19, 9-21
- Asn1Type class
 - functions for parsing Morf instances, 9-12
 - get_bit_string_identifiers function, 9-16, 9-17 to 9-18
 - get_enum_identifiers function, 9-16 to 9-18
 - get_range function, 9-18 to 9-21
 - get_size_constraint function, 9-21 to 9-23
 - Morf class and, 9-15
- Asn1Value
 - printing, 10-28
- Asn1Value class
 - decoding data, functions for, 9-30
 - getting from Morf instance, 9-28 to 9-29
 - strings, transformation by Morf class, 9-2
- assigning nicknames to managed objects, 5-11 to 5-12
- asynchronous operations
 - introduction, 8-2
 - cancelling, 8-24
 - CMIS, 8-5 to 8-10
 - completion of
 - callback code, 8-14 to 8-15
 - callback registration, 8-11 to 8-12
 - error handling, 8-23 to 8-24
 - on managed objects, 8-3

- MIS, interaction with, 8-3
 - on object collections, 8-4, 8-5 to 8-10
 - response handling, 8-11 to 8-23
 - return values, 8-2
 - testing for completion of, 8-21
 - timeouts, changing, 8-25
- atomic synchronization, 6-21
- ATTR_CHANGED event, 7-5
- attribute
 - AdministrativeState, 10-13
 - code generation and filters, 10-13
 - DiscriminatorConstruct, 10-13
 - OperationalState, 10-13
- Attribute GDMO template
 - definition, C-11 to C-13
 - example, 2-11
- Attribute Group GDMO template, C-20 to C-21
- attribute value assertion (AVA), 2-18
- attributes
 - defining, 2-9 to 2-11
 - denial of access to, 12-35
 - failure to set, 15-23 to 15-25
 - filters, in, 6-9, 6-10
 - getting, 5-17 to 5-18, 8-7
 - management operations permitted, 2-20
 - names in function calls, checking, 15-19 to 15-20
 - operations on
 - Image class, 5-21
 - in asynchronous CMIS set request, 8-9
 - read-only, modifying, 5-31 to 5-32
 - retrieving from MDR, 5-30
 - setting
 - asynchronous CMIS M_SET request, 8-7 to 8-10
 - Image class, using, 5-19 to 5-23
 - in object collections, 6-19 to 6-20
- ATTRIBUTES construct
 - getting attributes, 5-17
 - operations on attributes, 5-21, 8-9
 - read-only attributes, 5-31
 - setting attributes, 5-19
 - values allowed for attributes, 5-20
- attributeValueChange event, 7-3, 7-5
- AttrModifier class, 8-8 to 8-10
- AuthFeatures class, 12-6, 12-7
- AUTOIMAGE property, Album class, 6-16
- automatic assignment, managed object names, 15-20
- auxiliary objects, event access control, 12-37, 12-38
- AuxServerUtils class, 12-44
- AVA (attribute value assertion), 2-18

B

- backing store, em_debug message types, 15-15
- base managed object
 - access control, 12-23
 - event filtering, 7-17
 - object collections, 6-6
- BASE_OBJECT scope value, 12-24
- BASE_TO_NTH_LEVEL scope value, 12-24
- baseToNthLevel scope value, 7-18
- basic encoding rules (BER), 15-8 to 15-9
- BEGIN keyword, ASN.1 modules, 2-23
- behavior code, 1-17 to 1-18
- behavior, defining in object model, 2-13
- Behaviour GDMO template, C-22
- BER (basic encoding rules), 15-8 to 15-9
- best effort synchronization, 6-21
- bibliography, B-5
- BIT STRING ASN.1 type
 - definition, 2-26
 - identifiers, getting, 9-16, 9-17 to 9-18
 - size constraints, getting, 9-21 to 9-23
 - string representation, Morf class, 9-25
- blocking user interaction, 7-14 to 7-15
- BOOLEAN ASN.1 type
 - definition, 2-26
 - creating Morf instances for, 9-5
 - string representation, Morf class, 9-25
- boot function, Image class, 5-4, 5-14
- brace notation
 - names of managed objects, 2-20
 - OIDs, 2-33 to 2-34
- built-in ASN.1 types, 2-26
- bulk operations, 6-1
- bypassing MISs, 3-5 to 3-10

C

- C shell, escape characters, 9-27
- call function, Image class, 5-24
- call_raw function, Image class, 5-24
- Callback class
 - asynchronous operations, 8-11 to 8-12
 - event handling, 7-5
 - secure MPAs, 12-43
- callback functions
 - adding to scheduler queue, 8-22 to 8-23
 - asynchronous operation completion, 8-14 to 8-15
 - example, 7-8, 8-18 to 8-20

- extracting event information, 7-7
- hi_process_ace_event, 12-43
- Image instances, updating, 5-27
- lo_process_ace_event, 12-43
- object collections, updating, 6-15 to 6-16
- registering
 - access control events, 12-43 to 12-44
 - asynchronous operations, 8-11 to 8-12
 - event handling, 7-4 to 7-6
 - responses from managed objects, 8-12 to 8-13
- responses from managed objects
 - extracting information from, 8-17 to 8-20
 - registration, 8-12 to 8-13
 - scheduler data, correct use of, 8-15 to 8-16
- signature
 - asynchronous operations, 8-14
 - event handling, 7-6
- tracking changes to managed objects, 7-10 to 7-11
- cancel function, Waiter class, 8-24
- case sensitivity, GDMO identifiers, 15-20
- cellSample example, 10-35
- CFLAGS makefile entry, 14-7
- chai example, 10-49
- CHOICE ASN.1 type
 - definition, 2-26
 - creating Morf instances for, 9-6
 - encoding, 9-35 to 9-37
 - extracting values from, 9-27 to 9-28
 - formatting string representation of, 9-27
 - parsing, 9-12 to 9-13
 - string representation, Morf class, 9-25
- className.load, 10-27, 10-32
- className.unload, 10-27, 10-32
- className_error, 10-29
- className_info, 10-29
- className_user.odt.cc, 10-31
- className_user.odt.hh, 10-30
- CMIP (Common Management Information Protocol)
 - em_debug message types, 15-15 to 15-16
 - support for, 2-3
 - translation, monitoring, 15-3 to 15-4
- CMIS (common management information service), 8-5
- code generator utility, 10-8
- CODEGENDIR, 10-13
- collections, *See* object collections
- command names
 - Actions menu, 16-7
 - Tools menu, 16-5
- command-line arguments
 - Actions menu applications, 16-7
 - Tools menu applications, 16-5
 - tools window applications, 16-3
 - topology type activation, 16-9
- Common Management Information Protocol (CMIP)
 - em_debug message types, 15-15 to 15-16
 - support for, 2-3
 - translation, monitoring, 15-3 to 15-4
- common management information service (CMIS), 8-5
- communications protocols, *See* management protocols
- communicationsAlarm event, 7-3
- comparison keywords, filters, 6-10
- compilation guidelines
 - applications, 14-1 to 14-8
 - debugging programs, 15-3
 - source code examples, A-1 to A-2
- compilers, 2-36
 - C++, 14-1
 - object model translation, 2-36
- components, of applications, 1-2 to 1-3
- concise MIB compiler, 2-36
- conditional packages, 2-20
- configuration
 - OCG, 10-13
 - OCG, CODEGENDIR, 10-13
 - OCG, DATASTORAGE, 10-13
 - OCG, FILTER_ATTR, 10-13
 - OCG, HIDDENDIR, 10-13
 - OCG, OBAPIDEBUG, 10-13
 - OCG, OBAPITRACE, 10-13
- configuration files
 - environment variables, 12-39
 - Network Views tool, 16-7 to 16-8, 16-9 to 16-10
 - Solstice EM tools, 16-4 to 16-5
 - tools windows, 16-2 to 16-3
- configuration parameters, 1-6
- connect function, Platform class, 3-3
- connecting
 - to databases, 3-5 to 3-10
 - to MISs
 - generally, 3-1 to 3-3
 - asynchronously, 8-3
- connection manager, em_debug message types, 15-16
- Connection target, 12-19
- connection to MIS, managed object representing, 7-23
- constraints, ASN.1 values, 9-18 to 9-23
- constructors, error checking, 4-2
- containment tree, defining, 2-14 to 2-20
- context-specific class ASN.1 tags, 15-9

- converting SNM 2.x schema files to GDMO descriptions, 17-5
- CREATE construct, 5-2
- create function
 - ACDbObject class, 12-37
 - ACGroup class, 12-11
 - ACRule class, 12-27
 - ACTargets class, 12-20
 - Image class, 5-6
- create operation, targets, 12-21
- create_within function, Image class, 8-3, 15-20
- creating
 - managed objects
 - generally, 5-2 to 5-8
 - in agent simulation, 5-32 to 5-35
 - object collections, container for, 6-2
- CurrentEvent class
 - introduction, 7-7
 - do_nothing function, 7-9
 - do_something function, 7-10
 - functions for extracting information
 - from events, 7-7
 - from responses, 8-17
- customizing
 - Actions menu, Network Views tool, 16-6 to 16-8
 - Network Views tool, 16-6 to 16-10
 - Solstice EM tools, 16-4 to 16-5
 - tools windows, 16-1 to 16-3
 - topology type activation, 16-9 to 16-10

D

- data
 - access control, *See* managed objects, access control
 - isolating from program code, 1-2
 - presenting to users, 1-8
- data component, of applications, 1-3
- data types, defining, 2-23 to 2-29
- databases
 - direct access to, 3-5 to 3-10
 - modifying, 15-5 to 15-6
- DATASTORAGE, 10-13
- deactivating
 - access control, 12-9 to 12-10
 - Image instances, 8-3, 8-4
- debugging
 - overview, 1-22
 - ASN.1 syntax and logic, 15-2

- attributes, failure to set, 15-23 to 15-25
- compilation flags for, 14-8
- event handling, 15-25 to 15-26
- filters, 15-20 to 15-23
- GDMO syntax and logic, 15-2
- high-level PMI calls, 15-2
- logic errors, 15-2 to 15-3
- managed objects, failure to create, 15-20
- management operations, 15-2
- MIS communications, 15-7 to 15-19
- name bindings and, 15-20
- names, in function calls, 15-19 to 15-20
- object model updates, 15-4 to 15-6
- protocol translation, 15-3 to 15-4
- scopes, 15-20 to 15-23
- debugging agents, 10-29
- debugging ASN.1, 10-28
- debugging GDMO, 10-28
- debugging objects, 10-26
- debugging port, em_debug message types, 15-16
- decoding ASN.1 values
 - introduction, 9-1
 - ENUMERATED values, 9-29
 - examples, A-8
 - lists, 9-27 to 9-28
 - OBJECT IDENTIFIER values, 9-29
 - scalars, 9-28 to 9-31
 - SEQUENCE values, 9-27 to 9-28
 - SET values, 9-27 to 9-28
 - as strings, 9-25 to 9-27
- decoding functions, Asn1Value class, 9-30
- DEFAULT operation, Image class, 5-21
- defining object behavior, 10-5
- definitions
 - protocol adaptors, 11-1
 - SAPs, 11-4
 - Service Access Points, 11-4
- DEFINITIONS keyword, ASN.1 modules, 2-23
- DELETE construct, 5-15
- delete operation, targets, 12-21
- DELETES-CONTAINED-OBJECTS modifier, 5-15
- deleting
 - GDMO documents from MDR, 15-4 to 15-6
 - managed objects
 - generally, 5-15 to 5-16
 - asynchronously, 8-10
 - in object collection, 6-18 to 6-19
- demoPing example, 10-36
- demoregistry example, 10-42

- demoServer example, 10-47
- denial granularity, access control, 12-35
- deny without response, 12-8
- DenyAccesscontrolObjectsChange access control rule, 12-26
- DenyAccessControlObjectsChange target, 12-19
- denyWithFalseResponse enforcement action, 12-30
- denyWithoutResponse enforcement action, 12-30
- denyWithResponse enforcement action, 12-30
- derivation strings
 - introduction, 6-5
 - base managed object, 6-6
 - examples, 6-11 to 6-12
 - filters
 - introduction, 6-7 to 6-8
 - attributes, 6-9, 6-10
 - comparison keywords, 6-10
 - items, 6-9
 - operators, 6-9
 - substrings, 6-10 to 6-11
 - scope, 6-6 to 6-7
 - setting, 6-3 to 6-4
- derivation, of object collections
 - introduction, 6-3
 - setting derivation string, 6-3 to 6-4
 - starting
 - generally, 6-5
 - asynchronously, 8-4
- derive function, Album class
 - generally, 6-5
 - restrictions, 6-15
- destroy function, Image class, 5-15
- development environment, uses of, 1-1
- development process
 - overview, 1-4
 - debugging, 1-22
 - examples, A-2 to A-3, A-8
 - high-level design, 1-4 to 1-13
 - implementation, 1-19 to 1-22
 - integration, 1-22
 - low-level design, 1-13 to 1-18
 - requirements analysis, 1-4 to 1-13
 - system testing, 1-22
 - unit testing, 1-22
- devices, 1-5 to 1-6, 1-7
- diagnostic information, providing to users, 4-3
- diagnostic messages, 15-2
- disconnect function, Platform class, 3-4
- DISCONNECTED event, 7-5
- disconnecting from MISs
 - generally, 3-4
 - asynchronously, 8-3
- discriminator construct, 7-19 to 7-20
- DiscriminatorConstruct, 10-13
- discriminators, em_debug message types, 15-16
- dispatch_main_loop function, 7-12 to 7-13
- dispatch_recursive function
 - event handling, 7-11 to 7-12
 - secure MPAs and, 12-42
- "DNFILTER" : emDnScope attribute, 7-18
- do_nothing function, CurrentEvent class, 7-9
- do_something function, CurrentEvent class, 7-10
- documents, GDMO, 2-22
- domains, access control
 - generally, 12-36
 - specifying for secure MPAs, 12-42
- dot notation, OIDs, 2-33
- double click behavior, topology types, 16-9 to 16-10
- duEM platform type, 3-2
- dynamic libraries, 10-27

E

- em_accesscmd utility, 12-5 to 12-6
- em_admintool.cf file, 16-2 to 16-3
- em_alarmmgr_tp.cf file, 16-4 to 16-5
- em_asn1 compiler, 2-36
- em_cmib2gdmo compiler, 2-36
- em_compose_all script, 2-38
- em_compose_oc command, 2-37 to 2-38
- em_compose_poc command, 2-37 to 2-38
- em_debug
 - ODT agents, 10-29
- em_debug utility
 - introduction, 15-7
 - message types, 15-8, 15-15 to 15-19
 - output from, 15-8 to 15-14
 - starting, 15-7 to 15-8
- em_gdmo compiler, 2-36
- EM_GOTOVIEW macro, 16-8, 16-10
- em_load_name_bindings command, 2-38
- EM_LOG_MPA_EVENT_ACCESS environment variable, 12-39
- em_logmgr_tp.cf file, 16-4 to 16-5
- em_logview_tp.cf file, 16-4 to 16-5
- em_nnadd command, 5-11
- em_nnconfig command, 5-12

- em_nnpa daemon, 5-10
- em_objop utility
 - generally, 5-32 to 5-35
 - examples, A-12
- em_panel.cf file, 16-2 to 16-3
- em_services command, 15-5, 15-6
- em_snm2gdmo compiler, 17-5
- EM_TARGETS target type, 12-19
- em_viewer.cf file, 16-4 to 16-5, 16-7 to 16-8, 16-9 to 16-10
- EM-config configuration file, 12-39
- EMDBConnectInfo class, 3-5 to 3-7
- emSpecialEvents attribute
 - log record events, 7-23
 - secure MPAs, 12-40
- encoding ASN.1 values
 - introduction, 9-1
 - ANY and ANY DEFINED BY values, 9-7 to 9-8
 - CHOICE values, 9-6, 9-35 to 9-37
 - examples, A-8
 - lists, 9-2 to 9-3
 - MorfBuilder class, by using, 9-32 to 9-39
 - scalars, 9-5
 - SEQUENCE and SEQUENCE OF values, 9-8 to 9-9, 9-37 to 9-38
 - SET and SET OF values, 9-8 to 9-9
 - from string data, 9-2 to 9-4
- END keyword
 - ASN.1 module, 2-23
 - GDMO documents, 2-22
- enforcement actions
 - defining, 12-29 to 12-30
 - events, 12-33 to 12-34
 - management operations, 12-33
- enhancing performance
 - generally, 13-1 to 13-6
 - event handling, 7-9
 - managed objects, 5-35 to 5-37
 - object collections, 6-15
- enq function, Queue class, 8-8
- ENUMERATED ASN.1 type
 - definition, 2-26
 - creating Morf instances for, 9-5
 - extracting values from, 9-29
 - identifiers, getting, 9-16 to 9-18
- enumeration, of object collections, 6-12
- environmentalAlarm event, 7-3
- equality keyword, filters, 6-10
- equipmentAlarm event, 7-3

- Error class
 - generally, 4-2 to 4-3
 - debugging, use in, 15-2
- error handling
 - access control
 - generally, 12-31 to 12-32
 - rule creation, 12-27
 - target creation, 12-20
 - in asynchronous operations, 8-23 to 8-24
 - in constructors, 4-2
 - device errors, 1-6
 - in function calls, 4-1 to 4-3
 - high-level PMI, 4-1 to 4-3
 - in synchronous operations, 4-1 to 4-3
 - user errors
 - generally, 1-9
 - preventing, 7-14 to 7-15
- error types, 4-2
- error.hh file, 4-2
- Event Logs tool, customizing, 16-4 to 16-5
- event notifications, *See* events; notifications
- events
 - introduction, 7-1
 - access control
 - introduction, 12-3
 - auxiliary object, 12-37, 12-38
 - enforcement actions, 12-33 to 12-34
 - log owners, assigning, 12-36 to 12-39
 - log server, enabling, 12-39
 - processing information in, 12-43 to 12-44
 - subscribing to, 12-40 to 12-41
 - callback function registration
 - generally, 7-4 to 7-6
 - secure MPAs, 12-43 to 12-44
 - defined by standard X.721, 7-3
 - em_debug message types, 15-16
 - examples, A-5 to A-6
 - extracting information from, 7-7
 - failure to process, 15-25 to 15-26
 - filtering
 - introduction, 7-16
 - by event type, 7-16 to 7-17
 - by managed object class, 7-16 to 7-17
 - selecting subtree of MIT, 7-17 to 7-19
 - specifying discriminator construct, 7-19 to 7-20
 - listening for, *See* events, scheduling; subscribing, to
 - log record events
 - log record, 7-23 to 7-24
 - managed objects associated with, getting, 7-7

- MISs associated with, getting, 7-7
- object collections associated with, getting, 7-7
- OIDs of, getting, 7-7
- performance considerations, 13-3
- pre-empting automatic updates, 7-10
- recognized by `when` function, 7-5
- scheduling
 - introduction, 7-11
 - customization guidelines, 7-15
 - graphical applications, 7-13 to 7-15
 - nongraphical applications, 7-11 to 7-13
- sending to MIS, 7-21
- simulating, 7-20 to 7-22
- standard, 7-3
- updating managed objects, 7-9 to 7-11
- X Window system, 7-13 to 7-15
- example
 - `cellSample`, 10-35
 - `chai`, 10-49
 - `demoPing`, 10-36
 - `demoregistry`, 10-42
 - `demoServer`, 10-47
 - ODT, 10-34
- examples
 - access control, A-9 to A-10
 - compilation guidelines, A-1 to A-2
 - development scenarios, A-2 to A-3, A-8
 - encoding and decoding ASN.1 values, A-8
 - event handling, A-5 to A-6
 - FDN translation, A-7
 - graphical applications, A-7
 - high-level PMI, A-3 to A-8
 - log record handling, A-6
 - low-level PMI, A-10
 - managed objects, A-3 to A-4
 - MDR, querying, A-8
 - miscellaneous, A-12
 - object collections, A-5
 - object modeling, A-11
 - ODT, A-11, A-12
 - topology, A-7
 - troubleshooting scenarios, 15-23 to 15-26
- exception macros, 10-32
- exceptions, `em_debug` message types, 15-16
- `ExceptionType` class, 8-23, 8-24
- `exclude` function, `Album` class, 6-15
- `EXCLUDE` operation, `Image` class, 5-21
- `exists` function
 - `ACDBObject` class, 12-37

- `ACGroup` class, 12-11
- `ACRule` class, 12-27
- `ACTargets` class, 12-20
- `Image` class, 5-5
- `EXPORTS` keyword, ASN.1 modules, 2-29
- extending
 - Actions menu, Network Views tool, 16-6 to 16-8
 - Tools menu, Solstice EM tools, 16-4 to 16-5
 - tools windows, 16-1 to 16-3
- `extract` function, `Morf` class, 9-11, 9-12, 9-27 to 9-28
- extracting ASN.1 values
 - from lists, 9-27 to 9-29
 - from scalars, 9-28 to 9-31
- F**
 - failed operations, reporting, 4-1
 - false response, 12-30
 - faults, *See* error handling
 - FDNs (fully distinguished names)
 - introduction, 2-18 to 2-19
 - of connection to MIS, getting, 7-23
 - of event sources, getting, 7-7
 - specifying in management requests, 5-9
 - translation, examples, A-7
 - features, controlling access to
 - introduction, 12-2
 - enforcing predefined rules, 12-7 to 12-8
 - getting feature list, 12-16 to 12-18
 - privilege groups, 12-18
 - file descriptors, 7-15
 - files
 - configuration
 - environment variables, 12-39
 - Network Views tool, 16-7 to 16-8, 16-9 to 16-10
 - Solstice EM tools, 16-4 to 16-5
 - tools windows, 16-2 to 16-3
 - files, MPA debug output, 15-4
 - filter attributes, 10-13
 - `filter` operation, targets, 12-22
 - `FILTER_ATTR`, 10-13
 - filtering events
 - introduction, 7-16
 - by event type, 7-16 to 7-17
 - by managed object class, 7-16 to 7-17
 - selecting subtree of MIT, 7-17 to 7-19
 - specifying discriminator construct, 7-19 to 7-20
 - filters

- introduction, 6-7 to 6-8
- access control, 12-24
- attributes, 6-9, 6-10
- comparison keywords, 6-10
- em_debug message types, 15-16
- items, 6-9
- operators, 6-9
- substrings, 6-10 to 6-11
- testing support for, 15-20 to 15-23
- finalString keyword, filters, 6-11
- first_album function, Image class, 6-23
- first_image function, Album class, 6-22
- flags, compilation
 - generally, 14-8
 - debugging tools and, 15-3
- format bits, Morf class, 9-26 to 9-27
- formatting
 - Morf instance
 - data, 9-3
 - string representation, 9-26 to 9-27
 - tools windows, 16-2
- Forte compilers, 15-3
- FROM keyword, ASN.1 modules, 2-28
- Full Access
 - privilege group, 12-10
 - rule, access control, 12-26
- fully distinguished names (FDNs)
 - introduction, 2-18 to 2-19
 - of connection to MIS, getting, 7-23
 - of event sources, getting, 7-7
 - specifying in management requests, 5-9
 - translation, examples, A-7
- function calls, error handling, 4-1 to 4-3
- further reading, B-5

G

- GDMO
 - sanity check, 10-6
- GDMO (Guidelines for the Definition of Managed Objects), 2-5
 - syntax and logic, verifying, 15-2
- GDMO compiler, 2-36
- GDMO compiler (em_gdmo), 17-5
- GDMO document, 17-5
- GDMO documents
 - introduction, 2-22
 - examples, A-11

- identifiers, case sensitivity, 15-20
- names, getting from MDR, 5-29
- reloading, 15-4 to 15-6
- GDMO packages
 - introduction, 2-20 to 2-22
 - retrieving from MDR, 5-30
- GDMO templates
 - Action, C-13 to C-15
 - Attribute
 - definition, C-11 to C-13
 - example, 2-11
 - Attribute Group, C-20 to C-21
 - Behaviour, C-22
 - conventions, C-1
 - Managed Object Class
 - definition, C-2 to C-3
 - example, 2-7 to 2-8
 - Name Binding
 - definition, C-4 to C-6
 - example, 2-16, 2-17 to 2-18
 - Notification
 - definition, C-15 to C-17
 - example, 2-12 to 2-13
 - Package
 - definition, C-7 to C-11
 - example, 2-21 to 2-22
 - Parameter, C-18 to C-20
- generated code, isolating from handwritten code, 1-3
- GenInt class, 9-29
- geographical maps, em_debug message types, 15-16
- get function, Morf class, 9-25 to 9-26
- get operation, targets, 12-21
- get_access_control_switch function, ACAccessControlRules class, 12-9 to 12-10
- get_access_user_list function, ACInterface class, 12-13
- get_album function, CurrentEvent class, 7-7, 8-17
- get_all_applications function, ACAApplicationContainer class, 12-16
- get_all_features function, ACAAppFeatureContainer class, 12-17
- get_application_container function, ACInterface class, 12-16
- get_application_description function, ACAApplication class, 12-16
- get_authorized_features function, Platform class, 12-6, 12-7
- get_bit_string_identifiers function, Asn1Type class, 9-12, 9-16, 9-17 to 9-18

- get_database_name function, EMDBConnectInfo class, 3-7
- get_dbl function
 - Image class, 5-17
 - Morf class, 9-29
- get_default_access function, ACAccessControlRules class, 12-33
- get_default_event_access function, ACAccessControlRules class, 12-33
- get_denial_granularity function, ACAccessControlRules class, 12-35
- get_domain_identity function, ACAccessControlRules class, 12-36
- get_enum_identifiers function, AsnlType class, 9-12, 9-16 to 9-18
- get_error_string function
 - access control API, 12-31
 - ACRule class, 12-27
 - ACTargets class, 12-20
 - Error class, 4-3, 15-2
- get_error_type function
 - access control API, 12-31
 - ACRule class, 12-27
 - ACTargets class, 12-20
 - Error class, 4-2 to 4-3, 15-2
- get_event function, CurrentEvent class, 7-7
- get_event_raw function, CurrentEvent class, 7-7
- get_eventtype function, CurrentEvent class, 7-7, 8-17
- get_except function, Waiter class, 8-23
- get_feature_description function, ACAppliationFeature class, 12-17
- get_gint function
 - Image class, 5-17
 - Morf class, 9-29
- get_image function, CurrentEvent class, 7-7, 8-17
- get_info function, CurrentEvent class, 7-7
- get_info_raw function, CurrentEvent class, 7-7, 8-17
- get_long function, Image class, 5-17
- get_member_name function, Morf class, 9-11, 9-15
- get_memname function, Morf class, 9-11, 9-12
- get_message function, CurrentEvent class, 7-7, 8-17
- get_name function, CurrentEvent class, 7-7
- get_nickname function, Image class, 5-13
- get_objclass function, CurrentEvent class, 7-7, 8-17
- get_objname function, CurrentEvent class, 7-7, 8-17
- get_oid function, CurrentEvent class, 7-7
- get_platform function, CurrentEvent class, 7-7
- get_platform function, Morf class, 9-11, 9-15
- get_prop function
 - MorfBuilder class, 9-37 to 9-38
 - Platform class, 7-23, 12-40
- get_range function, AsnlType class, 9-12, 9-18 to 9-21
- get_raw function
 - Image class, 5-17
 - MorfBuilder class, 9-33, 9-39
- get_role function, EMDBConnectInfo class, 3-7
- get_server_name function, EMDBConnectInfo class, 3-7
- get_server_type function, EMDBConnectInfo class, 3-7
- get_set_dbl function, Image class, 5-22
- get_set_gint function, Image class, 5-22
- get_set_long function, Image class, 5-22
- get_set_raw function, Image class, 5-22
- get_set_str function, Image class, 5-22
- get_size_constraint function, AsnlType class, 9-12, 9-21 to 9-23
- get_status function, EMDBConnectInfo class, 3-7
- get_str function
 - Image class, 5-17
 - Morf class, 9-29
- get_syntax function, Morf class, 9-11, 9-15
- get_time function, CurrentEvent class, 7-7
- get_trusted_host_list function, ACAccessControlRules class, 12-34
- get_type function, Morf class, 9-11, 9-15, 9-16, 9-30
- get_user_name function, EMDBConnectInfo class, 3-7
- get_user_password function, EMDBConnectInfo class, 3-7
- get_value function, Morf class, 9-31
- getting
 - attributes, 5-17 to 5-18
 - asynchronously, 8-7
 - metadata, actions for, 5-29 to 5-30
 - value in last set request, 5-22
- glyph files, default location, 16-2
- granularity, denial of requests, 12-35
- grapher API
 - introduction, 1-17
 - examples, A-12
- Grapher tool, 1-17

- graphical applications, examples, A-7
- graphical user interface (GUI) component, 1-3
- greaterOrEqual keyword, filters, 6-10
- grouping managed objects, 6-1 to 6-2
- groups, *See* privilege groups
- GUI (graphical user interface) component, 1-3
- Guidelines for the Definition of Managed Objects (GDMO), 2-5
 - syntax and logic, verifying, 15-2

H

- Handler block, 10-32
- has_value function, Morf class, 9-11, 9-16
- header files, 14-1 to 14-7
- hi_process_ace_event callback function, 12-43
- HIDDENDIR, 10-13
- hierarchy
 - managed objects, 2-14
 - managers and agents, 2-4
- high-level design, 1-4 to 1-13
- high-level PMI
 - introduction, 1-15
 - debugging calls to, 15-2
 - em_debug message types, 15-17
 - error handling, 4-1 to 4-3
 - performance considerations, 13-1
- historical data, object collections, 6-14, 6-15
- how to generate code, 10-12

I

- icons, in tools windows, 16-2
- IGNORE operation, Image class, 5-21
- Image class
 - introduction, 5-2
 - activating instances of
 - generally, 5-4
 - asynchronously, 8-3
 - in object collection, 6-16 to 6-17, 8-4
 - asynchronous functions, 8-3
 - boot function, 5-4, 5-14
 - call function, 5-24
 - call_raw function, 5-24
 - constructor, 5-3
 - create function, 5-6
 - create_within function, 8-3, 15-20
 - deactivating instances of, 8-3, 8-4
 - DEFAULT operation, 5-21
 - destroy function, 5-15
 - EXCLUDE operation, 5-21
 - exists function, 5-5
 - find_by_nickname function, 5-13
 - first_album function, 6-23
 - get_dbl function, 5-17
 - get_gint function, 5-17
 - get_long function, 5-17
 - get_nickname function, 5-13
 - get_raw function, 5-17
 - get_set_dbl function, 5-22
 - get_set_gint function, 5-22
 - get_set_long function, 5-22
 - get_set_raw function, 5-22
 - get_set_str function, 5-22
 - get_str function, 5-17
 - IGNORE operation, 5-21
 - imaginary values, 5-22 to 5-23
 - INCLUDE operation, 5-21
 - instance associated with event, getting, 7-7
 - operations on attributes, 5-21
 - performance considerations, 13-2, 13-3
 - real values, 5-22 to 5-23
 - REPLACE operation, 5-21
 - restrictions, 5-35
 - send_event function, 7-21
 - set_dbl function, 5-19
 - set_gint function, 5-19
 - set_long function, 5-19
 - set_nickname function, 5-13
 - set_prop function, 5-26
 - set_raw function, 5-19
 - set_str function, 5-19
 - shutdown function, 8-3
 - start functions, 8-3
 - start_create function, 8-23
 - store function, 5-21
 - TRACKMODE property, 5-26, 5-27
 - updating instances of
 - asynchronously, 8-3
 - in response to application requests, 5-14
 - in response to network activity, 5-26 to 5-27
 - when function, 7-4 to 7-5
 - IMAGE_EXCLUDED event, 7-5
 - IMAGE_INCLUDED event, 7-5
 - imaginary values, 5-22 to 5-23
 - implementation, overview, 1-19 to 1-22

- IMPORTS keyword, ASN.1 modules, 2-28
- include files, *See* header files
- include function, Album class, 6-15
- INCLUDE operation, Image class, 5-21
- individualLevels scope value, 7-18
- information, *See* data; management information
- inheritance, managed object classes, 2-8
- initialization, em_debug message types, 15-17
- initializing, managed objects, 5-5 to 5-6
- initialString keyword, filters, 6-11
- insert function
 - ACMOCList defined type, 12-25
 - ACMOList defined type, 12-23
 - ACOperationsList defined type, 12-21 to 12-22
- INTEGER ASN.1 type
 - definition, 2-26
 - creating Morf instances for, 9-5
 - ranges, getting, 9-18 to 9-21
- integer values
 - checking value last set, 5-22
 - getting, 5-17
 - setting
 - in managed objects, 5-19
 - in object collections, 6-19
- integrating, applications, 1-22, 16-1 to 16-10
- integrityViolation event, 7-3
- International Organization for Standardization (ISO)
 - network management model, 1-2, 2-1 to 2-4
- is_any function, Morf class, 9-11
- is_authorized function, AuthFeatures class, 12-7
- is_choice function, Morf class, 9-11, 9-12
- is_list function, Morf class, 9-11, 9-13
- is_sequence function, Morf class, 9-11, 9-13
- is_set function, Morf class, 9-11, 9-13
- ISO (International Organization for Standardization)
 - network management model, 1-2, 2-1 to 2-4
- ISO registration tree, 2-30 to 2-31
- ITU-T X.208/ISO-8824 *Specification of Abstract Syntax Notation One (ASN.1)*, 2-23
- ITU-T X.209/ISO-8825 *Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1)*, 15-8
- ITU-T X.710/ISO-9595 *Common Management Information Services (CMISE)*, 1-14, 8-5
- ITU-T X.721/ISO-10165-2 *Definition of Management Information*, 7-3
- ITU-T X.722/ISO-10165-4 *Guidelines for the Definition of Managed Objects (GDMO)*, 2-5
- ITU-T X.735/ISO 10164-6 *Log Control Function*, 12-37
- ITU-T X.741 *Objects and Attributes for Access Control*, 12-19

L

- label texts
 - Actions menu commands, 16-7
 - Tools menu commands, 16-5
 - tools windows, 16-3
- launcher, *See* tools windows
- layout, icons in tools windows, 16-2
- LDFLAGS makefile entry, 14-7
- LDLIBS makefile entry, 14-1 to 14-7
- LDNs (local distinguished names)
 - introduction, 2-19
 - specifying in management requests, 5-9
- lessOrEqual keyword, filters, 6-10
- libraries, 14-1 to 14-7
 - manager/agent services API, 17-1
- limitations, verifying for managed resources, 2-7
- limits, ASN.1 values, 9-18 to 9-23
- linking applications, guidelines for, 14-1 to 14-8
- listening for events, *See* scheduling, event handling;
 - subscribing, to log record events
- lists
 - ASN.1 type of, getting, 9-13
 - creating Morf instances for, 9-2 to 9-3
 - decoding, 9-27 to 9-28
 - length of, getting, 9-13
 - members, getting ASN.1 types of, 9-15
 - parsing, 9-13 to 9-15
 - splitting, 9-14 to 9-15
- lo_process_ace_event callback function, 12-43
- Load Data Definitions tool, 2-36, 2-37
- loading libraries, 10-27
- local distinguished names (LDNs)
 - introduction, 2-19
 - specifying in management requests, 5-9
- local objects, 1-17
- local root, 2-19
- locating managed objects, 2-14
- location, header files and libraries, 14-7
- log class, 12-37
- Log Entries tool, customizing, 16-4 to 16-5
- log management, em_debug message types, 15-17
- log server, enabling access control, 12-39
- LOG_SVC defined type, 3-6
- logic errors, 15-2 to 15-3
- logs

- direct access to, 3-5
- events from
 - examples, A-6
 - subscribing to, 7-23 to 7-24
- owners, assigning to, 12-36 to 12-39
- long integers
 - checking value last set, 5-22
 - getting, 5-17
 - setting
 - in managed objects, 5-19
 - in object collections, 6-19
- low-level design, 1-13 to 1-18
- low-level PMI
 - introduction, 1-14
 - examples, A-10
 - performance enhancement, use in, 13-3 to 13-6

M

- macros, 10-32
- maintainability, of applications, 1-2
- makefile, entries required by Solstice EM, 14-1 to 14-7
- Managed Object Class GDMO template
 - definition, C-2 to C-3
 - example, 2-7 to 2-8
- managed object classes
 - abstracting, 2-7 to 2-8
 - adding to MIS, 2-37 to 2-38
 - definitions, retrieving from MDR, 5-29
 - of event sources, getting, 7-7
 - filtering events by, 7-16 to 7-17
 - inheritance, 2-8
 - names in function calls, verifying, 15-19 to 15-20
 - selecting all instances of, 12-24 to 12-25
- managed objects
 - introduction, 1-2, 2-3
 - access by applications
 - generally, 3-1 to 3-3
 - asynchronous, 8-3
 - access control
 - introduction, 12-3
 - denial, handling, 12-8
 - granularity, of denial, 12-35
 - operations permitted, defining, 12-21 to 12-22
 - rules, 12-29, 12-33
 - selecting objects for, 12-18 to 12-21, 12-23 to 12-25
 - actions, performing
 - generally, 5-23 to 5-25

- asynchronously, 8-3, 8-4
 - on object collections, 6-20
- adding to MIS
 - asynchronously, 8-3, 8-4
 - in object collections, 6-18
 - individually, 5-6 to 5-7
- attributes
 - defining, 2-9 to 2-11
 - denial of access to, 12-35
 - failure to set, 15-23 to 15-25
 - filters, in, 6-9, 6-10
 - getting, 5-17 to 5-18, 8-7
 - management operations permitted, 2-20
 - names in function calls, checking, 15-19 to 15-20
 - operations on, 5-21, 8-9, 12-22
 - read-only, modifying, 5-31 to 5-32
 - retrieving from MDR, 5-30
 - setting, 5-19 to 5-23, 6-19 to 6-20, 8-7 to 8-10
- C++ representation, 5-35 to 5-37
- creating
 - generally, 5-2 to 5-8
 - in agent simulation, 5-32 to 5-35
- deleting
 - generally, 5-15 to 5-16
 - asynchronously, 8-10
 - in object collection, 6-18 to 6-19
- designing object model
 - overview, 2-5
 - actions, 2-11
 - attributes, 2-9 to 2-11
 - behavior, 2-13
 - classes, 2-7 to 2-8
 - containment, 2-14 to 2-20
 - documents, 2-22
 - examples, A-11
 - inheritance, 2-8
 - management operations, 2-6 to 2-7
 - notifications, 2-12 to 2-13
 - packages, 2-20 to 2-22
- direct containment by MIS, 5-31
- enhancing performance, 5-35 to 5-37
- examples, A-3 to A-4
- existence of, verifying, 5-5
- finding all object collections for, 6-23 to 6-24
- grouping, 6-1 to 6-2
- hierarchy, 2-14
- initializing, 5-5 to 5-6
- multiple selection, 6-3 to 6-12
- names

- automatic assignment, 15-20
- brace notation, 2-20
- fully distinguished, 2-18 to 2-19
- local distinguished, 2-19
- relative distinguished, 2-18
- nicknames of, assigning, 5-11 to 5-12
- pre-empting event-related updates, 7-10
- removing from MIS
 - generally, 5-15 to 5-16
 - asynchronously, 8-3
- representing connection to MIS, getting, 7-23
- selecting
 - all instances of a class, 12-24 to 12-25
 - by FDN or LDN, 5-9, 12-23
 - by nickname, 5-10 to 5-13
 - from object collections, 6-21 to 6-22
 - subtree of MIT, 6-5 to 6-12, 8-6 to 8-7, 12-23 to 12-24
- tracking changes to
 - automatically, 7-9
 - from callback functions, 7-10 to 7-11
 - Image class, using, 5-26 to 5-27
 - manually, 7-10
 - overriding updates, 7-9
- managed resources
 - introduction, 2-3
 - capabilities, verifying, 2-7
- management information
 - presenting to users, 1-8
 - sharing, 1-9 to 1-10
- management information base (MIB), 2-36
- management information servers (MISs)
 - introduction, 3-1
 - agent role behavior, 2-37
 - auxiliary objects
 - adding, 12-37
 - storing persistently, 12-38
 - bypassing, 3-5 to 3-10
 - communications with
 - limiting, 13-1 to 13-6
 - monitoring, 15-7 to 15-19
 - connecting to
 - generally, 3-1 to 3-3
 - asynchronously, 8-3
 - containing managed objects in, 5-31
 - current state of, verifying, 5-21 to 5-22
 - disconnecting from
 - generally, 3-4
 - asynchronously, 8-3
 - of event sources, getting, 7-7
 - events, sending to, 7-21
 - managed object classes, loading, 2-37 to 2-38
 - managed object representing connection to, 7-23
 - managed objects
 - adding, 5-6 to 5-7, 8-3
 - removing, 5-15 to 5-16, 8-3
 - managing several, 1-12 to 1-13
 - name bindings, loading, 2-38
 - nickname service, adding, 5-10 to 5-11
 - object collections
 - adding, 6-18, 8-4
 - removing, 6-18 to 6-19, 8-4
 - privilege groups
 - adding, 12-11
 - storing persistently, 12-14 to 12-15
 - purging, 15-5
 - rules, access control
 - adding, 12-27
 - storing persistently, 12-30 to 12-31
 - starting, 15-5, 15-6
 - stopping, 15-5
 - targets, access control
 - adding, 12-20
 - storing persistently, 12-25
 - updating after set request
 - asynchronously, 8-3, 8-4
 - on managed object, 5-21
 - on object collection, 6-20
 - users, adding, 12-13 to 12-14
- management information tree (MIT)
 - introduction, 2-14
 - direct manipulation, 15-2
 - managed objects, selecting, 6-3
 - subtree, selecting
 - access control, 12-23 to 12-24
 - asynchronous CMIS operations, 8-6 to 8-7
 - event filtering, 7-17 to 7-19
 - object collections, 6-5 to 6-12
- management operations
 - access control, 12-21 to 12-22
 - CMIS, 8-5
 - debugging, 15-2
 - defining, 2-6 to 2-7
 - denial granularity, 12-35
 - enforcement actions, 12-29 to 12-30, 12-33
 - permitted on attributes, 2-20
 - supported by Solstice EM, 5-2
 - synchronization, 6-21

- management protocol adapters (MPAs)
 - access control
 - introduction, 12-3
 - ACE class, 12-42 to 12-43
 - AuxServerUtils class, 12-44
 - connection to MIS, 12-40
 - decision and enforcement functions, 12-44
 - domains, 12-42
 - events, subscribing to, 12-40 to 12-41
 - processing information in events, 12-43 to 12-44
 - services required, 12-42 to 12-43
 - comparison with using ODT, 1-18
 - debug mode, 15-3 to 15-4
 - filters, support for, 15-20 to 15-23
 - scopes, support for, 15-20 to 15-23
 - starting, 15-3 to 15-4
 - stopping, 15-4
- management protocols, 2-3 to 2-4
- manager-agent model
 - agent, 17-1
- managers
 - introduction, 2-2
 - hierarchical arrangement, 2-4
 - in ISO model, 1-2
- mandatory packages, 2-20
- MDR (metadata repository)
 - actions, 5-29 to 5-30
 - ASN.1 types, representation of, 9-2
 - em_debug message types, 15-17
 - naming attribute, 5-28
 - populating, 2-36
 - querying
 - generally, 5-27 to 5-30
 - examples, A-8
 - updating, 15-4 to 15-6
- memory leaks, avoiding, 8-15 to 8-16
- memory, saving, 5-35, 13-1 to 13-6
- message routing module, em_debug message types, 15-17
- messages
 - See also* notifications, 2-12
 - from devices, 1-5 to 1-6
- metadata repository (MDR)
 - actions, 5-29 to 5-30
 - ASN.1 types, representation of, 9-2
 - em_debug message types, 15-17
 - naming attribute, 5-28
 - populating, 2-36
 - querying
 - generally, 5-27 to 5-30
 - examples, A-8
 - updating, 15-4 to 15-6
- metadata, effect of disconnection on, 3-4
- metaName attribute, MDR managed object, 5-28
- MIB (management information base), 2-36
- MIS Objects tool, 5-32, 15-2
- MISs (management information servers)
 - introduction, 3-1
 - agent role behavior, 2-37
 - auxiliary objects
 - adding, 12-37
 - storing persistently, 12-38
 - bypassing, 3-5 to 3-10
 - communications with
 - limiting, 13-1 to 13-6
 - monitoring, 15-7 to 15-19
 - connecting to
 - generally, 3-1 to 3-3
 - asynchronously, 8-3
 - containing managed objects in, 5-31
 - current state of, verifying, 5-21 to 5-22
 - disconnecting from
 - generally, 3-4
 - asynchronously, 8-3
 - of event sources, getting, 7-7
 - events, sending to, 7-21
 - managed object classes, loading, 2-37 to 2-38
 - managed object representing connection to, 7-23
 - managed objects
 - adding, 5-6 to 5-7, 8-3
 - removing, 5-15 to 5-16, 8-3
 - managing several, 1-12 to 1-13
 - name bindings, loading, 2-38
 - nickname service, adding, 5-10 to 5-11
 - object collections
 - adding, 6-18, 8-4
 - removing, 6-18 to 6-19, 8-4
 - privilege groups
 - adding, 12-11
 - storing persistently, 12-14 to 12-15
 - purging, 15-5
 - rules, access control
 - adding, 12-27
 - storing persistently, 12-30 to 12-31
 - starting, 15-5, 15-6
 - stopping, 15-5
 - targets, access control
 - adding, 12-20

- storing persistently, 12-25
 - updating after set request
 - asynchronously, 8-3, 8-4
 - on managed object, 5-21
 - on object collection, 6-20
 - users, adding, 12-13 to 12-14
- MIT (management information tree)
 - introduction, 2-14
 - direct manipulation, 15-2
 - managed objects, selecting, 6-3
 - subtree, selecting
 - access control, 12-23 to 12-24
 - asynchronous CMIS operations, 8-6 to 8-7
 - event filtering, 7-17 to 7-19
 - object collections, 6-5 to 6-12
- modes, of object collections, 6-16 to 6-17
- modification list, 8-8
- modifying, Solstice EM database, 15-5 to 15-6
- MODULE keyword, GDMO documents, 2-22
- monitoring
 - MIS communications, 15-7 to 15-19
 - protocol translation, 15-3 to 15-4
- Morf class
 - introduction, 9-1 to 9-2
 - Asn1Type instance associated with, 9-15
 - constructors
 - other Morf instances, 9-9
 - string data, 9-2
 - creating instances from MorfBuilder instance, 9-39
 - default representation, ASN.1 types, 9-25
 - examples, A-8
 - extract function, 9-11, 9-12, 9-27 to 9-28
 - format bits, 9-26 to 9-27
 - get function, 9-25 to 9-26
 - get_dbl function, 9-29
 - get_gint function, 9-29
 - get_member_name function, 9-11, 9-15
 - get_memname function, 9-11, 9-12
 - get_platform function, 9-11, 9-15
 - get_str function, 9-29
 - get_syntax function, 9-11, 9-15
 - get_type function, 9-11, 9-15, 9-16, 9-30
 - get_value function, 9-31
 - has_value function, 9-11, 9-16
 - is_any function, 9-11
 - is_choice function, 9-11, 9-12
 - is_list function, 9-11, 9-13
 - is_sequence function, 9-11, 9-13
 - is_set function, 9-11, 9-13
 - lists, representation by, 9-2 to 9-3
 - metainformation, 9-16 to 9-24
 - navigation strings, 9-12, 9-27 to 9-28
 - num_elements function, 9-11, 9-13
 - Platform instance associated with, 9-15
 - set function, 9-5
 - set_any function, 9-7
 - set_dbl function, 9-5
 - set_gint function, 9-5
 - set_long function, 9-5
 - set_memname function, 9-6
 - set_str function, 9-5
 - split_array function, 9-11, 9-14
 - split_queue function, 9-11, 9-14
 - Syntax instance associated with, 9-15
- MorfBuilder class
 - introduction, 9-32
 - access_type property, 9-37 to 9-38
 - constructors, 9-33
 - creating Morf instances from, 9-39
 - data, adding to instance of, 9-33 to 9-35
 - get_prop function, 9-37 to 9-38
 - get_raw function, 9-33, 9-39
 - navigation strings, 9-34, 9-37
 - set function, 9-33 to 9-35
 - set_prop function, 9-37 to 9-38
 - set_raw function, 9-33 to 9-34
 - validate function, 9-33, 9-38 to 9-39
- MPAs (management protocol adapters)
 - access control
 - introduction, 12-3
 - ACE class, 12-42 to 12-43
 - AuxServerUtils class, 12-44
 - connection to MIS, 12-40
 - decision and enforcement functions, 12-44
 - domains, 12-42
 - events, subscribing to, 12-40 to 12-41
 - processing information in events, 12-43 to 12-44
 - services required, 12-42 to 12-43
 - comparison with using ODT, 1-18
 - debug mode, 15-3 to 15-4
 - filters, support for, 15-20 to 15-23
 - scopes, support for, 15-20 to 15-23
 - starting, 15-3 to 15-4
 - stopping, 15-4
 - multiple selection, of managed objects, 6-3 to 6-12
 - multipleObjectSelection operation, targets, 12-22

multi-valued attributes, 5-21, 8-9

N

Name Binding GDMO template

- definition, C-4 to C-6

- example, 2-16, 2-17 to 2-18

name bindings

- introduction, 2-15

- adding to MIS, 2-38

- containment, defining, 2-15

- creation, managed objects, 2-15, 5-2

- deletion, managed objects, 2-15, 5-15

- example

 - multiple levels, 2-17 to 2-18

 - one level, 2-16

- multiple containment levels, 2-16 to 2-18

name function, `ExceptionType` class, 8-24

names

- managed objects

 - automatic assignment, 15-20

 - brace notation, 2-20

 - fully distinguished, 2-18 to 2-19

 - local distinguished, 2-19

 - relative distinguished, 2-18

- object model items, retrieving from MDR, 5-29

naming attribute

- definition, 2-15

- MDR managed object, 5-28

navigation strings

- `Morf` class, 9-12, 9-27 to 9-28

- `MorfBuilder` class, 9-34, 9-37

Nerve Center interface

- introduction, 1-15

- `em_debug` message types, 15-17 to 15-18

- examples, A-12

network management model, of Solstice EM, 2-1 to 2-4

network resources, *See* managed resources

Network Tools window, customizing, 16-2 to 16-3

Network Views tool

- topology API and, 1-15

- viewer API and, 1-16

Network Views tool, customizing, 16-4 to 16-5, 16-6 to 16-10

networks, 1-7

`next_album` function, `AlbumImage` class, 6-23

`next_image` function, `AlbumImage` class, 6-22

nickname service

- introduction, 5-10

- adding nicknames to, 5-12

- adding to MIS, 5-10 to 5-11

- starting, 5-10

nicknames

- introduction, 5-10

- adding to nickname service, 5-12

- Album instances, 6-2

- assigning to managed objects, 5-11 to 5-12

- examples, A-12

- getting, 5-13

- Image instance, finding, 5-13

- setting, 5-13

`nonNullSetIntersection` keyword, filters, 6-10

`not` keyword, filters, 6-9

`NOT` operator, 4-2

Notification GDMO template

- definition, C-15 to C-17

- example, 2-12 to 2-13

notifications

- introduction, 2-2

- defining, 2-12 to 2-13

- failure to process, 15-25 to 15-26

- from devices, 1-5

- retrieving from MDR, 5-30

`NTH_LEVEL` scope value, 12-24

`NULL` ASN.1 type, definition, 2-26

`num_elements` function, `Morf` class, 9-11, 9-13

O

OAM (object access module), `em_debug` message types, 15-18 to 15-19

OBAPIDEBUG, 10-13

OBAPITRACE, 10-13

object

- debugging process, 10-26

- ODT interfaces, 10-3

object access module (OAM), `em_debug` message types, 15-18 to 15-19

object API, 10-4

Object Code Generator

- See* OCG

object collections

- introduction, 6-1

- actions, performing

 - generally, 6-20

 - asynchronously, 8-4, 8-10

- activating Image instances in
 - generally, 6-16 to 6-17
 - asynchronously, 8-4
- adding to MIS
 - generally, 6-18
 - asynchronously, 8-4
- attributes
 - getting, 8-7
 - setting, 6-19 to 6-20, 8-7 to 8-10
- base managed object, 6-6
- callback functions for, 6-15 to 6-16
- creating container for, 6-2
- deleting
 - generally, 6-18 to 6-19
 - asynchronously, 8-4
- em_debug message types, 15-19
- examples, A-5
- historical data, 6-14, 6-15
- individual objects, selecting, 6-21 to 6-22
- members
 - choosing, 6-1 to 6-2
 - deriving, 6-3 to 6-5, 8-4
 - enumerating, 6-12
 - tracking, 6-23 to 6-24
- MIS, updating after set request
 - generally, 6-20
 - asynchronously, 8-4
- modes of, 6-16 to 6-17
- performance considerations, 13-3 to 13-6
- setting attributes, 6-19 to 6-20
- subset of, selecting, 8-6 to 8-7
- synchronization, 6-21
- timeouts, 6-20
- tracking changes, 6-13 to 6-17
- updating, 6-13 to 6-17
- Object Development Tools
 - See ODT
- object development tools (ODT)
 - comparison with writing MPAs, 1-18
 - examples, A-11, A-12
 - restrictions, 1-18
- object framework, 10-4
- OBJECT_IDENTIFIER ASN.1 type
 - definition, 2-26
 - extracting values from, 9-29
 - formatting string representation of, 9-27
- object identifiers (OIDs)
 - allocating, guidelines for, 2-32 to 2-33
 - brace notation, 2-33 to 2-34
 - dot notation, 2-33
 - of events, getting, 7-7
 - labelling, 2-34
 - of Solstice EM, 2-30
 - registering, 2-30 to 2-31
 - retrieving from MDR, 5-29
- object model
 - introduction, 2-1
 - designing
 - overview, 2-5
 - actions, 2-11
 - attributes, 2-9 to 2-11
 - behavior, 2-13
 - classes, 2-7 to 2-8
 - containment, 2-14 to 2-20
 - documents, 2-22
 - inheritance, 2-8
 - management operations, 2-6 to 2-7
 - notifications, 2-12 to 2-13
 - packages, 2-20 to 2-22
 - examples, A-11
 - loading into MDR, 2-36
 - SNMP MIBs, 2-36
 - standards as basis for, 2-35
 - updating, 15-4 to 15-6
- object services, 10-4
- object services API, 1-15
- object utilities, 10-4
- OBJECT_CREATED event, 7-5
- OBJECT_DESTROYED event, 7-5
- objectCreation event, 7-3, 7-5
- objectDeletion event, 7-3, 7-5
- objects
 - See also managed objects
 - behavior code, 1-17 to 1-18
 - location, 1-17
- objsvc_error, 10-29
- objsvc_test, 10-29
- OBSAPI, 10-4
- OCG, 10-8
 - className.load, 10-32
 - className.unload, 10-32
 - className_user.odt.cc, 10-31
 - className_user.odt.hh, 10-30
 - client create file, 10-31
 - CODEGENDIR, 10-13
 - components, 10-10
 - configuration, 10-13
 - DATASTORAGE, 10-13

- dynamic loading file, 10-32
 - dynamic unloading file, 10-32
 - exception-handling macros, 10-32
 - filter attributes, 10-13
 - `FILTER_ATTR`, 10-13
 - generated code interfaces, 10-9
 - generated files, 10-29
 - `HIDDENDIR`, 10-13
 - how to use, 10-12
 - inputs, 10-11
 - `Makefile`, 10-30
 - `OBAPIDEBUG`, 10-13
 - `OBAPITRACE`, 10-13
 - outputs, 10-11
 - `pmi_className.cc`, 10-31
 - `README`, 10-30
 - user code file, 10-31
 - user header file, 10-30
 - `OCTET STRING` ASN.1 type
 - definition, 2-26
 - creating `Morf` instances for, 9-5
 - formatting string representation of, 9-27
 - size constraints, getting, 9-21 to 9-23
 - string representation, `Morf` class, 9-25
 - ODT
 - `cellSample` example, 10-35
 - `chai` example, 10-49
 - `className.load`, 10-27
 - `className.unload`, 10-27
 - code generation components, 10-10
 - components, 10-3
 - definition, 10-1
 - `demoPing` example, 10-36
 - `demoregistry` example, 10-42
 - `demoServer` example, 10-47
 - dynamic libraries, 10-27
 - `em_debug` agents, 10-29
 - examples, 10-34
 - generated code, 10-9
 - interfaces, 10-3
 - Object Behavior Interface, 10-4
 - Object Code Generator, 10-8
 - object development scenario, 10-49
 - operations, 10-2
 - process, 10-5
 - sanity check, 10-6
 - ODT (object development tools)
 - comparison with writing MPAs, 1-18
 - examples, A-11, A-12
 - restrictions, 1-18
 - `Oid` class, 9-29
 - OIDs (object identifiers)
 - allocating, guidelines for, 2-32 to 2-33
 - brace notation, 2-33 to 2-34
 - dot notation, 2-33
 - of events, getting, 7-7
 - labelling, 2-34
 - of Solstice EM, 2-30
 - registering, 2-30 to 2-31
 - retrieving from MDR, 5-29
 - Open Systems Interconnection (OSI), standards supported by Solstice EM, B-1 to B-2
 - `OperationalState`, 10-13
 - `operationalViolation` event, 7-3
 - operations on attributes
 - access control, 12-22
 - asynchronous CMIS set request, 8-9
 - `Image` class, 5-21
 - operator overloading, 4-2
 - Operators privilege group, 12-10
 - `or` keyword, filters, 6-9
 - OSAPI
 - debug agents, 10-29
 - OSI (Open Systems Interconnection), standards supported by Solstice EM, B-1 to B-2
 - overloaded operators, 4-2
 - overriding automatic updates to managed objects, 7-9
- ## P
- Package GDMO template
 - definition, C-7 to C-11
 - example, 2-21 to 2-22
 - packages, GDMO
 - introduction, 2-20 to 2-22
 - retrieving from MDR, 5-30
 - Parameter GDMO template, C-18 to C-20
 - parameters
 - OCG, `CODEGENDIR`, 10-13
 - OCG, `DATASTORAGE`, 10-13
 - OCG, `FILTER_ATTR`, 10-13
 - OCG, `HIDDENDIR`, 10-13
 - OCG, `OBAPIDEBUG`, 10-13
 - OCG, `OBAPITRACE`, 10-13
 - parameters, configuration, 1-6
 - parsing ASN.1 values
 - introduction, 9-9 to 9-10

- BIT STRING values, 9-16, 9-17 to 9-18, 9-21 to 9-23
- CHOICE values, 9-12 to 9-13
- ENUMERATED values, 9-16 to 9-18
- example, 9-23 to 9-24
- lists, 9-13 to 9-15
- OCTET STRING values, 9-21 to 9-23
- REAL values, 9-18 to 9-21
- SEQUENCE OF values, 9-21 to 9-23
- SET OF values, 9-21 to 9-23
- type, getting, 9-16
- performance
 - devices, of, 1-6
 - enhancing
 - generally, 13-1 to 13-6
 - event handling, 7-9
 - managed objects, 5-35 to 5-37
 - object collections, 6-15
 - networks, of, 1-7
- performance, enhancing, 5-21
- physicalViolation event, 7-3
- Platform class
 - introduction, 3-1
 - connect function, 3-3
 - constructor, 3-2
 - disconnect function, 3-4
 - get_authorized_features function, 12-6, 12-7
 - get_prop function, 7-23, 12-40
 - Morf class and, 9-15
 - replace_discriminator function, 7-17 to 7-18, 7-19
 - replace_discriminator_classes function, 7-16 to 7-17
 - restrictions, 3-2
 - start_connect function, 8-3
 - start_disconnect function, 8-3
 - when function, 7-4 to 7-5
- platform types, 3-2
- PMI (Portable Management Interface)
 - high-level
 - introduction, 1-15
 - debugging calls to, 15-2
 - em_debug message types, 15-17
 - error handling, 4-1 to 4-3
 - examples, A-3 to A-8
 - performance considerations, 13-1
 - low-level
 - introduction, 1-14
 - examples, A-10
 - performance enhancement, use in, 13-3 to 13-6
- pmi_className.cc, 10-31
- polling, devices, 1-5, 1-7
- Portable Management Interface (PMI)
 - high-level
 - introduction, 1-15
 - debugging calls to, 15-2
 - em_debug message types, 15-17
 - error handling, 4-1 to 4-3
 - examples, A-3 to A-8
 - performance considerations, 13-1
 - low-level
 - introduction, 1-14
 - examples, A-10
 - performance enhancement, use in, 13-3 to 13-6
- pre-empting automatic updates, events, 7-10
- present keyword, filters, 6-10
- presenting management information, 1-8
- private class ASN.1 tags, 15-9
- privilege groups
 - introduction, 12-10
 - applications, adding, 12-18
 - creating, 12-11 to 12-12
 - features, adding, 12-18
 - MIS, adding to, 12-11
 - predefined, 12-10
 - rules, adding to, 12-28 to 12-29
 - storing persistently, 12-14 to 12-15
 - users, adding, 12-14 to 12-15
- process
 - defining object behavior, 10-5
- processingErrorAlarm event, 7-3
- programming model, 1-2 to 1-3
- property list, ATTRIBUTES construct
 - definition, C-9 to C-11
 - getting attributes, 5-17
 - operations on attributes, 5-21, 8-9
 - read-only attributes, 5-31
 - setting attributes, 5-19
 - values allowed for attributes, 5-20
- protocol adaptors, 11-1
- protocol translation, debugging, 15-3 to 15-4
- protocols, *See* management protocols
- proxy agent
 - manage resources not directly accessible, 17-2
 - protocol translation, 17-2
- purging, Solstice EM database, 15-5

Q

- quality assurance, *See* system testing; unit testing
- qualityofServiceAlarm event, 7-3
- Queue class, 8-8
 - asynchronous CMIS M-SET request, 8-8
 - Morf instance, use in creating, 9-8

R

- RAW_EVENT event, 7-5
- RDNs (relative distinguished names), 2-18
- read-only attributes, 5-31 to 5-32
- REAL ASN.1 type
 - definition, 2-26
 - creating Morf instances for, 9-5
 - ranges, getting, 9-18 to 9-21
- real values
 - checking value last set, 5-22
 - getting, 5-17
 - setting
 - in managed objects, 5-19
 - in object collections, 6-19
- real values in Image instance, 5-22 to 5-23
- reason function, ExceptionType class, 8-24
- rebuilding, Solstice EM database, 15-5
- recommended books, B-5
- registering callback functions
 - access control events, 12-43 to 12-44
 - asynchronous operations, 8-11 to 8-12
 - event handling, 7-4 to 7-6
 - responses from managed objects, 8-12 to 8-13
- registering OIDs, 2-30 to 2-31
- relationshipChangeEvent, 7-3
- relative distinguished names (RDNs), 2-18
- remote objects, 1-17
- remote procedure call (RPC) protocol
 - support for, 2-3
 - translation, monitoring, 15-3 to 15-4
- REMOVE operation, asynchronous CMIS M-SET request, 8-9
- removeMember operation, targets, 12-22
- removing, GDMO documents from MDR, 15-4 to 15-6
- REPLACE operation
 - asynchronous CMIS M-SET request, 8-9
 - Image class, 5-21
- replace operation, targets, 12-21
- replace_discriminator function, Platform class, 7-17 to 7-18, 7-19

- replace_discriminator_classes function, Platform class, 7-16 to 7-17
- replacewithDefault operation, targets, 12-22
- request templates, 7-21
- requests
 - introduction, 2-2
 - denial of, 12-35
- requirements analysis, 1-4 to 1-13
- reset_error function, access control API, 12-31
- responses
 - introduction, 2-2
 - callback functions
 - code, 8-15 to 8-20
 - registration, 8-12 to 8-13
 - extracting information from, 8-17 to 8-20
 - false, 12-30
 - scheduling handling of
 - introduction, 8-20
 - blocking mode, 8-21 to 8-22
 - nonblocking mode, 8-21
- restrictions
 - access control, activating, 12-9
 - asynchronous CMIS operations, 8-5
 - attributes
 - getting, 5-17
 - in multiple GDMO documents, 5-17 to 5-18, 5-20
 - setting, 5-19
 - compilers, A-2
 - derive function, Album class, 6-15
 - EM_GOTOVIEW macro, 16-8, 16-10
 - Error class, 4-3
 - GDMO documents, 2-22
 - get_error_type function, Error class, 4-1
 - Image class, 5-35
 - log owners, changing, 12-38
 - managed objects
 - creation, 5-2
 - deletion, 5-15
 - inheritance, 2-8
 - modifying Solstice EM database, 15-5
 - Morf instance, updating, 9-3
 - ODT, 1-18
 - Platform class, 3-2
 - reloading GDMO documents, 15-4
 - send_event function, Image class, 7-21
 - source code examples, A-1
 - start_create function, Image class, 8-23
 - start_m_action function, Album class, 8-10
 - start_m_action_raw function, Album class, 8-10

- start_m_set function, Album class, 8-8
- robustness, of applications, 1-2
- root object, 2-14
- RPC (remote procedure call) protocol
 - support for, 2-3
 - translation, monitoring, 15-3 to 15-4
- rules, access control
 - introduction, 12-26
 - creating, 12-26 to 12-28
 - defining, 12-4 to 12-6
 - enforcement actions
 - defining, 12-29 to 12-30
 - getting, 12-33
 - enforcing, 12-4, 12-6 to 12-8
 - MIS, adding to, 12-27
 - predefined, 12-26
 - privilege groups, adding, 12-28 to 12-29
 - storing persistently, 12-30 to 12-31
 - targets, adding, 12-29

S

- S98ipmpa script, 15-3
- sanity checking, 10-6
- SAPs, 11-4
- saving memory, 5-35, 13-1 to 13-6
- scalars
 - creating Morf instances for, 9-5
 - decoding, 9-28 to 9-31
- scenario
 - developing object behaviors, 10-49
- sched scheduler
 - em_debug message types, 15-19
 - events
 - receiving, 7-11 to 7-12
 - simulating, 7-21
 - responses to asynchronous operations, 8-20
- schedulers, correct use of data passed by, 8-15 to 8-16
- scheduling
 - callback function execution, 8-22 to 8-23
 - event handling
 - introduction, 7-11
 - customization guidelines, 7-15
 - graphical applications, 7-13 to 7-15
 - nongraphical applications, 7-11 to 7-13
 - response handling
 - introduction, 8-20
 - blocking mode, 8-21 to 8-22
 - nonblocking mode, 8-21
- Schema compiler (em_snm2gdm), 17-5
- scopes
 - access control, 12-23 to 12-24
 - event filtering, 7-17 to 7-18
 - object collections, 6-6 to 6-7
 - testing support for, 15-20 to 15-23
- Security tool, 12-5
- security, *See* access control
- securityServiceOrMechanismViolation event, 7-3
- select() system call, 7-15
- send_event function, Image class, 7-21
- send_resp function, Waiter class, 8-22
- sending events, to MIS, 7-21
- SEQUENCE ASN.1 type
 - definition, 2-26
 - creating Morf instances for, 9-2, 9-8 to 9-9
 - encoding, 9-37 to 9-38
 - extracting values from, 9-27 to 9-28
 - string representation, Morf class, 9-25
- SEQUENCE OF ASN.1 type
 - definition, 2-26
 - creating Morf instances for, 9-2, 9-8 to 9-9
 - size constraints, getting, 9-21 to 9-23
- Service Access Points, 11-4
- SET ASN.1 type
 - definition, 2-26
 - creating Morf instances for, 9-2, 9-8 to 9-9
 - extracting values from, 9-27 to 9-28
 - string representation, Morf class, 9-25
- set function
 - Morf class, 9-5
 - MorfBuilder class, 9-33 to 9-35
- SET OF ASN.1 type
 - definition, 2-26
 - creating Morf instances for, 9-2, 9-8 to 9-9
 - size constraints, getting, 9-21 to 9-23
- set_access_control_switch function, ACAccessControlRules class, 12-9 to 12-10
- set_any function, Morf class, 9-7
- set_app_context function, xtsched scheduler, 7-13
- set_auxobject_owner function, ACDBObject class, 12-38
- set_dbl function
 - Morf class, 9-5
- set_dbl function, Image class, 5-19
- set_derivation function, Album class, 6-3

- set_enforcement_action function, ACRule class, 12-30
- set_filter function, ACTargets class, 12-24
- set_gint function
 - Image class, 5-19
 - Morf class, 9-5
- set_long function
 - Image class, 5-19
 - Morf class, 9-5
- set_memname function, Morf class, 9-6
- set_moc_list function, ACTargets class, 12-25
- set_moi_list function, ACTargets class, 12-23
- set_nickname function, Image class, 5-13
- set_operations_list function, ACTargets class, 12-22
- set_operator function, AttrModifier class, 8-8 to 8-9
- set_prop function
 - Album class, 6-14, 6-17, 6-21
 - Image class, 5-26
 - MorfBuilder class, 9-37 to 9-38
- set_raw function
 - Image class, 5-19
 - MorfBuilder class, 9-33 to 9-34
- set_scope function, ACTargets class, 12-24
- set_str function
 - Image class, 5-19
 - Morf class, 9-5
- SET_TO_DEFAULT operation, asynchronous CMIS M-SET request, 8-9
- set_value function, AttrModifier class, 8-8
- set_X_event_processing function, xtsched scheduler, 7-14 to 7-15
- setting
 - attributes
 - asynchronously, 8-7 to 8-10
 - in managed objects, 5-19 to 5-23
 - in object collections, 6-19 to 6-20
 - derivation strings, 6-3 to 6-4
- shared libraries, *See* libraries
- sharing management information, 1-9 to 1-10
- shutdown function, Image class, 8-3
- shutdown manager, em_debug message types, 15-19
- signature, callback functions
 - asynchronous operations, 8-14
 - event handling, 7-6
- Simple Network Management Protocol (SNMP)
 - MIB as object model, 2-36
 - support for, 2-3
 - translation, monitoring, 15-3 to 15-4
- simulating
 - agents, 5-30 to 5-35
 - events, 7-20 to 7-22
- SNMP (Simple Network Management Protocol)
 - MIB as object model, 2-36
 - support for, 2-3
 - translation, monitoring, 15-3 to 15-4
- software abstraction, 2-3
- Solstice EM API component of applications, 1-2 to 1-3
- Solstice EM tools, customizing, 16-4 to 16-5
- source code examples
 - access control, A-9 to A-10
 - compilation guidelines, A-1 to A-2
 - development scenarios, A-2 to A-3, A-8
 - encoding and decoding ASN.1 values, A-8
 - event handling, A-5 to A-6
 - FDN translation, A-7
 - graphical applications, A-7
 - high-level PMI, A-3 to A-8
 - log record handling, A-6
 - low-level PMI, A-10
 - managed objects, A-3 to A-4
 - MDR, querying, A-8
 - miscellaneous, A-12
 - object collections, A-5
 - object modeling, A-11
 - ODT, A-11, A-12
 - topology, A-7
- split_array function, Morf class, 9-11, 9-14
- split_queue function, Morf class, 9-11, 9-14
- standards
 - supported by Solstice EM, B-1 to B-2
 - terminology references, B-3 to B-4
- start functions, Image class, 8-3
- start_connect function, Platform class, 8-3
- start_create function, Image class, 8-23
- start_derive function, Album class, 8-4
- start_disconnect function, Platform class, 8-3
- start_m_action function, Album class, 8-10
- start_m_action_raw function, Album class, 8-10
- start_m_get function, Album class, 8-7
- start_m_set function, Album class, 8-7 to 8-10
- starting
 - applications
 - overview, 1-8 to 1-9
 - from Actions menu, Network Views tool, 16-6 to 16-8
 - by double clicking topology nodes, 16-9 to 16-10

- from Tools menu, Solstice EM tools, 16-4 to 16-5
 - from tools windows, 16-1 to 16-3
- MISs, 15-5, 15-6
- MPAs, 15-3 to 15-4
- state information, 2-9
- stateChange event, 7-3
- stopping
 - MISs, 15-5
 - MPAs, 15-4
- store function
 - ACAccessUserList class, 12-13
 - ACDBObject class, 12-38
 - ACGroup class, 12-15
 - ACRule class, 12-30
 - ACTargets class, 12-25
 - Image class, 5-21
- strings, creating Morf instances from, 9-2 to 9-4
- string-valued attribute
 - checking value last set, 5-22
 - getting, 5-17
 - setting, 6-19
- string-valued attributes
 - setting, 5-19
- subclasses, 2-8
- subordinate object, 2-14
- subscribing
 - to access control events, 12-40 to 12-41
 - to log record events, 7-23 to 7-24
- subsetOf keyword, filters, 6-10
- substrings, filters, 6-10 to 6-11
- superclasses, 2-8
- superior object, 2-14
- supersetOf keyword, filters, 6-10
- synchronization
 - asynchronous CMIS requests
 - M-ACTION, 8-10
 - M-SET, 8-8
 - management operations, object collections, 6-21
- synchronous operations, 8-1
- Syntax class
 - introduction, 9-2
 - Morf class and, 9-15
- system object, 2-14
- system testing, 1-22
- systems management functions, supported by Solstice EM, B-2

T

- tag length value (TLV) encoding, 15-8 to 15-9
- tags, ASN.1, 15-9 to 15-10
- targets, access control
 - introduction, 12-18 to 12-19
 - creating, 12-19 to 12-21
 - operations permitted, defining, 12-21 to 12-22
 - rules, adding to, 12-29
- Telecommunications Management Network (TMN),
 - standards supported by Solstice EM, B-1
- templates, *See* GDMO templates
- terminology, references to standards documents, B-3 to B-4
- testing, *See* system testing; unit testing
- text labels
 - Actions menu commands, 16-7
 - Tools menu commands, 16-5
 - tools windows, 16-3
- time, of event, getting, 7-7
- timeDomainViolation event, 7-3
- timeouts
 - actions, object collections, 6-20
 - asynchronous operations, 8-25
 - denial without response, 12-8
- TLV (tag length value) encoding, 15-8 to 15-9
- TMN (Telecommunications Management Network),
 - standards supported by Solstice EM, B-1
- Tools menu, customizing, 16-4 to 16-5
- tools windows, customizing, 16-1 to 16-3
- top object class, 2-8
- topology API
 - introduction, 1-15
 - em_debug message types, 15-19
 - examples, A-7
- topology types, activation, customizing, 16-9 to 16-10
- tracing code, 14-8, 15-2 to 15-3
- tracking changes
 - from callback functions, 7-10 to 7-11
 - managed objects, 5-26 to 5-27
 - object collections, 6-13 to 6-17
- TRACKMODE property
 - Album class, 6-14
 - Image class, 5-26, 5-27
- traps, from devices, 1-5
- troubleshooting, 15-1 to 15-26
- trusted hosts, 12-34
- TRY block, 10-32
- TRY execption macros, 10-32
- tuning, applications, 13-1 to 13-2

U

- UDP (user datagram protocol), 15-3
- unit testing, 1-22, 15-1 to 15-26
- universal ASN.1 types
 - introduction, 2-26
 - class tags, 15-9
- Unknown attribute message, 15-19
- Unknown object class error message, 15-19
- unsolicited messages
 - See also* notifications
 - from devices, 1-5 to 1-6
- updating
 - Image instances
 - asynchronously, 8-3
 - in response to application requests, 5-14
 - in response to network activity, 5-26 to 5-27
 - object collections, 6-13 to 6-17
 - object model, 15-4 to 15-6
- user datagram protocol (UDP), 15-3
- user groups, *See* privilege groups
- users
 - access control, requirements, 12-10
 - creating, for access control, 12-12
 - error information for, 4-1 to 4-3
 - interaction with applications, 1-7 to 1-9
 - MIS, adding to, 12-13 to 12-14
 - privilege groups, adding to, 12-14 to 12-15
 - rules, access control, 12-28 to 12-29
- uses, of Solstice EM APIs, 1-1

V

- validate function, MorfBuilder class, 9-33, 9-38 to 9-39
- values, in filters, 6-9, 6-10
- version requirements, compilers, 14-1, A-2
- View Only
 - privilege group, 12-10
 - rule, access control, 12-26
 - target, 12-19
- viewer API
 - introduction, 1-16
 - examples, A-12

W

- WAIT event, 7-5

Waiter class

- introduction, 8-2
- cancel function, 8-24
- get_except function, 8-23
- send_resp function, 8-22
- was_completed function, 8-21
- when_resp function, 8-12 to 8-13, 8-22

waitmore function, Waiter class, 8-25

was_completed function, Waiter class, 8-21

when function, 7-4 to 7-5

when_resp function, Waiter class, 8-12 to 8-13, 8-22

X

- X Window system, 7-13 to 7-15
- X741_TARGETS target type, 12-19
- XtAppMainLoop function, xtsched scheduler, 7-13
- xtsched scheduler
 - introduction, 7-13
 - activating, 7-13 to 7-14
 - blocking user interaction, 7-14 to 7-15
 - event simulation, 7-21
 - initializing, 7-13 to 7-14
 - responses to asynchronous operations, 8-20
 - XtAppMainLoop function, 7-13