Solstice Enterprise Manager API Syntax Manual

Release 2.0

2550 Garcia Avenue
Mountain View, CA 94043
U.S.A.

SunSoft
A Sun Microsystems, Inc. Business

Please
Recycle

Adobe PostScript

Contents

Figures

Tables

Preface

The *Solstice Enterprise Manager API Syntax Manual* provides an extensive list of the classes and member methods (functions) defined in the Portable Management Interface (PMI) used to communicate with the Solstice Enterprise Manager™, hereafter referred to as Solstice EM™ (or EM) Management Information Server (MIS).

Who Should Use This Book

This document is intended for programmers developing applications to run on top of the Solstice EM MIS. You should be very familiar with C++ and have experience using complex programmatic interfaces.

Before You Read This Book

If you have just acquired the product, you should read the Overview chapter in the *Solstice Enterprise Manager Reference Manual* to learn about basic product features, functions, and components. You should also read the *Solstice Enterprise Manager Release Notes* for information on installing and starting, compatibility and minimum machine and software requirements, known problems, an inventory of the product components, and the latest information about the Solstice EM product.

The *Solstice Enterprise Manager Application Development Guide* should be consulted for examples on using the PMI when writing applications.

How This Book Is Organized

This document is organized as follows:

Chapter 1, “Application Programming Interface,” describes the application programming interfaces (APIs) available for use with the Viewer and Grapher applications.

Chapter 2, “Common API Classes,” describes the classes, methods, and data members that are useful when using the low- or high-level portions of the PMI.

Chapter 3, “High-Level PMI,” describes the classes and methods that you use for the high-level use of the PMI.

Chapter 4, “Low-Level PMI,” describes the classes and methods that you use for the low-level PMI.

Chapter 5, “Nerve Center Interface Library” describes the classes and methods that make up the Request Interface Library, which allows your application to build, send, and receive requests.

Chapter 6, “Object Services API,” explains how to use the object behavior services with the object behavior API.

Chapter 7, “Topology API,” explains how to use this tool to create applications without learning the details of the MIT naming tree.

Conventions Used in This Book

This section describes the conventions used in this book.

What Typographic Changes and Symbols Mean

The following table describes the type changes and symbols 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 `.login` file. Use `ls -a` to list all files. `system% You have mail.`

`AaBbCc123`	What you type, contrasted with on-screen computer output	`system%` **`su`**
		`Password:`

<AaBbCc123>	Command-line placeholder: replace with a real name or value	To delete a file, type **`rm`** *<filename>*.
AaBbCc123	Book titles, new words or terms, or words to be emphasized	Read Chapter 6 in *User's Guide*. These are called *class* options. You *must* be root to do this.

Shell Prompts in Command Examples

All command line examples in this guide use the C-shell environment. If you use either the Bourne or Korn shells, refer to `sh`(1) and `ksh`(1) man pages for command equivalents to the C-shell. 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	`machine_name%`
C shell superuser prompt	`machine_name#`
Bourne shell and Korn shell prompt	`$`
Bourne shell and Korn shell superuser prompt	`#`

Application Programming Interface 1

1.1 Introduction

This chapter describes how communication can be achieved with application programming interfaces (APIs). The APIs discussed are Enterprise Manager's Viewer, Grapher, and Application-to-Application APIs.

This chapter includes the following classes:

Table 1-1 Application Programming Interface Classes

Classes	**Application**	**Description**
AppInstComm Class	*Application-to-Application API*	Used to establish communication with the platform which communicates with another EM application
AppInstObj Class	*Application-to-Application API*	Used to identify the target application for sending message
AppRequest Class	*Application-to-Application API*	Used to request a single message
AppTarget Class	*Application-to-Application API*	Used to send messages to applications

Table 1-1 Application Programming Interface Classes

Classes	**Application**	**Description**
EMdataset Class	*Grapher API*	Stores values or attributes
EMdynamicDataset Class	*Grapher API*	Stores attributes of a dynamic dataset
EMgraph Class	*Grapher API*	Creates new graphs
EMStaticDataset Class	*Grapher API*	Stores values for graphing statically
Err Class	*Grapher API*	Provides error checking
ViewerAPI Class	*Viewer API*	Used to communicate with applications running the EM Viewer application

1.2 Viewer API

The Viewer API allows applications to communicate with and modify the EM Viewer. This allows platform developers to leverage both Viewer functionality and to integrate their applications with the Viewer. The Viewer, therefore, can serve as an application's central place for performing management tasks.

The Viewer API allows the application to do two things:

- Communicate with and modify the Viewer. For example, an application can get the Viewer's current view, or set the contents of the Viewer's footer.
- Register with the Viewer to receive Viewer-generated events. This involves telling the Viewer to send events, and, on the application's end, registering callbacks for selected viewer events and responding to them.

The Viewer Api has only one class: `viewerApi,` in AppInstComm, from the Application-to-Application API.

1.2.1 `ViewerAPI` *Class*

The following includes the member functions for the ViewerAPI class.

1.2.1.1 Member Functions

Communication is achieved by invoking actions on application instance objects, or *targets*, corresponding to one or more running applications. Individual applications are represented as the `AppInstObj`, which is a member of `AppTarget`.

viewerapi_send_request

```
static DU ViewerApi::viewerapi_send_request(
    AppInstObj &obj,
     ViewerApiAction action,
     DU &action_data
)
```

This function sends a synchronous confirmed request to a specified application object and returns with data. Zero is returned if there is an error. *action* is the operation for the Viewer to perform and *action_data* consists of all parameters to be passed to the Viewer. For example, if viewerSetCurrentZoomLevel is the *action* and *action_data* is `20` the effect is to set the Viewer’s zoom level to 20%.

viewer_start_send_request

```
static AppRequest
ViewerApi::viewerapi_start_send_request(
    AppInstObj &obj,
     ViewerApiAction action,
     DU &action_data
)
```

This function sends an asynchronous confirmed request to a specified *target* and returns handles for waiting. *action* is the operation for the Viewer to perform and *action_data* contains all parameters to be passed to the Viewer.

viewerapi_send_request_unconfirmed

```
static Result
ViewerApi::viewerapi_send_request_unconfirmed(
    AppInstObj &obj,
     ViewerApiAction action,
     DU &action_data
)
```

This function sends an unconfirmed request to a specified *target*, while *action* is the operation for the Viewer to perform. *action_data* contains all parameters to be passed to the Viewer.

viewerapi_build_target

```
static AppTarget
ViewerApi::viewerapi_build_target(
    Array(DU) &userid = Array(DU)()
)
```

This function builds the *target* set of viewer applications to whom api actions are sent. *userid* is the system ID of the person running the application.

viewerapi_send_request

```
static Array (AppRequest)
ViewerApi::viewerapi_send_request(
    AppTarget &target,
    ViewerApiAction action,
    DU &action_data
)
```

This function sends a synchronous confirmed request to one or more *targets* and returns with data. Zero is returned if there is an error. *action* is the operation for the Viewer to perform and *action_data* contains all parameters to be passed to the Viewer.

viewerapi_start_send_request

```
static Array(AppRequest)
ViewerApi::viewerapi_start_send_request(
    AppTarget &target,
    ViewerApiAction action,
    DU &action_data
)
```

This function sends an asynchronous confirmed request to one or more *targets* and returns handles for waiting. *action* is the operation for the Viewer to perform and *action_data* contains all parameters to be passed to the Viewer.

viewerapi_send_request_unconfirmed

```
static Result
ViewerApi::viewerapi_send_request_unconfirmed(
    AppTarget &target,
    ViewerApiAction action,
    DU &action_data
)
```

This function sends an unconfirmed request to one or more *targets*. *action* is the operation for the Viewer to perform and *action_data* contains all parameters to be passed to the Viewer.

set_indication_handler

```
static AppEventHandler
    ViewerApi::set_indication_handler(
    AppEventHandler callback,
    const DU &pmt
)
```

This function registers a specific callback function (handler) *callback* for the event specified by *pmt*, defined as one of the GDMO PARAMETER templates shown in Code Example 1-1 on page 1-32 and Code Example 1-2 on page 1-33.

1.2.2 Communication Protocol

Two modes of communication (with an `em_viewer` application) are supported: *confirmed* and *unconfirmed.*

In the confirmed mode, the sender of the request waits for confirmation of receipt from the receiver. After confirmation is received, control is returned to the sender. The sender can also pass data to the receiver with the request, and the receiver can pass data with the response. Both synchronous and asynchronous versions are provided for confirmed requests.

In the unconfirmed mode, the request is sent to the receiving Viewer instance(s). The sender of the request neither requires nor waits for a confirmation of receipt.

Communication with an `em_viewer` application occurs in a specific order:

1. A program using `ViewerApi` sends a request to an `em_viewer`.
2. The receiving instance(s) of the Viewer get the notification and perform the action.
3. The receiving Viewer instance sends a response to the sending program.
4. The sending program receives confirmation.

1.2.2.1 Actions

An application written using `ViewerApi` class can communicate with any running viewer applications. All the actions supported by the Viewer API are described by enum ViewerApiAction. The `ViewerApi` class is declared in `viewer_api.hh.`

Note – The viewerSetViewCriteria action is equivalent to selecting a status name in View->Properties->Icon Status Color Source ->User-Defined. You can also pass the argument "SEVERITY" to select Alarm Severity instead of User-Defined.

The User-Defined option lets you set the icon color based on values that are set in topoNodeDisplayStatus attribute of topology nodes. This attribute contains a list of tag, value pairs, e.g. {{"CPU", 4 }, {"Diskload", 3}}. But to set this up you have to enable the tags by adding them to the topoAllStatus attribute of “topoNodeDB=NULL” object, e.g. {"CPU", "Diskload"} using OBED. The same color mapping used for severities is used for the user-defined values, so that means the values must be between 0 and 5. Additional color mappings can be added using OBED to expand the range of values, though.

For any action requested, each viewer application receiver responds with a reply. Table 1-2 summarizes defined actions.

Table 1-2 ViewerAPI Class Defined Actions

Viewer API Action	**Function**	**Request Data Format**	**Reply Data Format**
viewerGetCurrentView	Returns current view.	N/A	quoted string
viewerSetCurrentView	Sets Viewer’s view.	quoted string	status (TRUE or FALSE)
viewerSetFocusObject	Selects object in Viewer.	quoted string	status
viewerSetZoomLevel	Sets zoom level for view.	int (%)	status
viewerSetFooterContents	Places a text string in the Viewer’s footer.	quoted string	status
viewerSetViewCriteria	Sets which status variable is being used for coloring objects.	quoted string	status (TRUE or FALSE)
Viewer PopupMessage	Displays a message box with the appropriate message in the Viewer.	quoted string	status (TRUE or FALSE)
ViewerPopupQuestion	Displays a message box with appropriate message in the Viewer. Sends response back to the application.	quoted string	quoted string—a comma-separated string with 1st parameter TRUE/FALSE and reset the button number clicked beginning from 0)
viewerRegisterForEvents	Registers interest in Viewer events. (See Section 1.2.3, “Event Handling.”)	N/A	status (TRUE or FALSE)

Note – As of Solstice Enterprise Manager release 2.0, the request data of the `viewerSetCurrentView` action expects a view name in the format of system:id. For example, “\"myhost:0\".“ For more information, see the *Troubleshooting Guide.*

This action requires the following parameters:

1. **Modality**: modal or modeless. [MODAL | MODELESS]
2. **Dialog-type:** warning, error, information or question:

   [WARNING | ERROR | INFO | QUESTION]

   The dialog-type action determines the look of the dialog, including the title and the icon cue used.
3. **Message-text:** the text of the message to be displayed in the dialog.

1.2.2.2 *Inputs*

The `viewerPopupQuestionDialog` requires the same three parameters as the `viewerPopupMessageDialog`. In addition, the following parameters are required:

1. **num-buttons:** The number of buttons in the dialog.
2. **button-labels:** A sequence of <*num-buttons*> text labels for the buttons, from left to right.
3. **default-button:** The position of the default button to activate if the user hits the return key in the dialog. The position is a 0-based integer with 0 the left-most button.

Note – A status of `TRUE` means that the action requested from the viewer completed successfully; a status of `FALSE` means that the action requested from the viewer did not complete successfully.

1.2.3 *Event Handling*

Applications cannot only affect the Viewer; they can also listen for Viewer events, and, through callbacks, respond to those events. In this way, an application can use the Viewer as a central place for network management.

The general scheme is as follows:

1. **The application tells the Viewer of the events of which it has an interest.** It does this with the `AppInstComm:set_indication_handler()` method, described in Section 1.2.1.1, “Member Functions.”

For example, assume that the user has written a callback, `obj_sel_cb`, to do something when a person selects an object in the Viewer. `set_indication_handler` takes the callback as its first argument, and the Viewer event as its second:

```
ViewerApi va;          // create ViewerApi object
va = ViewrApi(em);     // em is the Platform
                       // notify Viewer we have a callback for
object selection
va.set_indication_handler(obj_sel_cb, duObjectSelectedEvent);
```

`ob_sel_cb` is called when a Viewer object is selected.

2. **The application notifies the Viewer that it is interested in events.** The application sets the ViewerAPIAction to viewerRegisterForEvents (described in Table 1-2), then calls `send_request_unconfirmed()` (Section 1.2.1.1, “Member Functions”). *Target* is the set of viewer applications to which API actions are sent.

```
//  set the action to send the viewer to 'register for events'
ViewerApiAction v_act = va.du2ViewerApiAction("viewerRegisterForEvent");
//  send the register action to the viewer
va.viewerapi_send_request_unconfirmed(target, v_act, DU()));
```

The syntax for `AppEventHandler:set_indication_handler()` is:

```
typedef int (IAppEventHander)
(
    DU input_info,
    DU &reply_action,
    DU &reply_info,
    AppInstObj sender
)
```

`viewerRegisterForEvents` notifies the Viewer that the application is interested in events *in general*; it becomes an On switch for event notification. The Viewer now informs the application of *all* events, not just the ones that the application has indicated an interest in (in Step 1). Of course, the application ignores all events except the ones for which it has callbacks.

3. The Viewer sends events to the application.

Table 1-3 shows the messages that the Viewer sends to applications when an event occurs. See Section 1.2.3.1, “Viewer Messages,” for more information.

Table 1-3 Viewer Event Messages

Viewer Message	**Data Format**	**Event Described**
duObjectSelectedEvent	quoted string	A user selected an object in the Viewer.
duObjectDeselectedEvent	quoted string	A user deselected an object in the Viewer.
duPopupMenuEvent	quoted string	A user brought up a Popup menu over an object.
duLayerChangeEvent	quoted string	The user moved a window onto another, changing the view.
duToolsMenuEvent	quoted string	The user selected from the Tools menu.
duObjectCreationEvent	quoted string	An object is added to the view being displayed
duObjectDeletionEvent	quoted string	An object is deselected from the current view
duViewChangeEvent	quoted string	The Viewer changes the current view
duRegisterForEvents	quoted string	Use this to tell the Viewer to send the calling application Viewer events

4. The registered callback for that event, if any, is called.

Note that an application must both

1. Notify the Viewer of specific events for which the application has callbacks, using `set_indication_handler()` (Step 1).

2. Notify the Viewer that it is interested in receiving events, using `ViewRegisterEvents` (Step 2).

Table 1-4 Using `ViewRegisterEvents`

Event Handlers	Interest in Receiving Event Notification Registered	Interest in Receiving Event Notification Not Registered
Event Handlers Registered	Viewer sends all events; registered callbacks called (non-registered callbacks ignored).	Viewer does not send event messages.
Event Handlers Not Registered	Viewer notifies of all events; all events ignored.	Viewer does not send event messages.

1.2.3.1 Viewer Messages

The following describes the Viewer messages shown in Table 1-3. These are messages sent to applications when a Viewer event occurs.

duObjectSelectedEvent

This message notifies the application that a user has selected an object in the viewer.

Input: This message is generated by any form of selection that the user performs in the Viewer (including adding an object to the current selection — that is, selecting a second object without deselecting the first).

Also, when the Viewer changes back to a previous view that contains already selected objects, messages are sent as if the user had just selected those objects.

Processing: Whenever the viewer performs a selection, it checks for any registered applications and sends them a message. If no applications have registered with the viewer, no message is generated (follows the format: `“system_name=unique_id”`).

Outputs: The action generated is viewerObjectSelection, and the parameter to the action is a quoted string containing the name(s) (format:*’system_name:unique_id’*) of the selected objects, separated by commas (for example, “*name:id1,name:id2,name:id3*”).

duObjectDeselected Event

This action informs applications when the user deselects objects in the Viewer.

Inputs: Any deselection action in the Viewer that causes objects to be deselected generates this message. This includes any new selection action that destroys the previous selection; it also includes deleting an object that is already selected. Additionally, it includes changing views. (The Viewer maintains selections across views, but the application listening for selection / deselection events doesn't need to do this.)

Processing: Whenever a deselection occurs, the viewer checks for any registered applications and sends them a message. If no applications have registered with the viewer, no message is generated.

Outputs: The action generated is viewerObjectDeselection. Its parameter is a quoted string containing a comma-separated list of the "*system_name:unique_ids*" of the objects being deselected.

duPopupMenuEvent

This event allows applications to be notified when the user brings up a popup menu item over an object.

Inputs: This message results from any selection, by a user, of an item on a popup menu in the Viewer canvas.

Processing: Whenever a popup menu is selected, the Viewer determines the object on which the popup menu was "popped," the menu item selected (the label, menu item number, and actual command associated with the popup) and generate a corresponding message. If no applications have registered with the viewer, no message is generated.

Outputs: The action generated is viewerPopupMenu. The parameter accompanying it is a quoted string containing a comma separated list as follows: '*topoNodeName*,*menu label*,*menu item #*,*menu command*'.

duLayerChange Event

This event notifies applications when a user changes the layers currently being displayed in a view (that is, moves a window on top of or underneath another window).

Inputs: Any change of the currently selected layers (including the background image) in the current view via the `Layers Dialog` produces this message. (Note: this does *not* include the various “levels” within a geographic map.) Any view change also generates a viewerLayerChange message reflecting the layer flags for the new view.

Processing: When any layer change is detected, the Viewer generates a message that formulates a list of all the active layers AND whether the background image is on / off and generates a message. If no applications have registered with the viewer, no message is generated.

Outputs: The action generated is viewerLayerChange. The parameter to the message is a quoted string containing a comma-separated list of all of the layers currently being displayed. The first entry in the list always represents the background image and is either “on” or “off” depending on the current value in the viewer.

duToolsMenuEvent

This event notifies applications when the user has selected something from the Tools Menu.

Inputs: The user makes a selection from the Tools menu.

Processing: When the user chooses something from the *Tools Menu*, the Viewer generates a message based on the user’s selection. If no applications have registered with the viewer, no message is generated

Outputs: The action generated is viewerToolsMenu. The parameter accompanying the message is a quoted string containing a comma-separated list as follows: ‘*menu label*,*menu item*,*menu command*’.

duObjectCreationEvent

This event notifies when any new object is added to the current view, either by the viewer or from an alternative source (such as discover). These objects include links.

Inputs: This message is generated when a user creates an object within a view using

- The Viewer GUI or

- Programmatic creation of objects within the view via another PMI program (such as Discover).

Processing: When an object is added to the view being displayed by the Viewer, the Viewer generates a message. If no applications have registered with the Viewer, no message is generated.

Outputs: The action generated is `viewerObjectCreationEvent`. The parameter accompanying the message is a quoted string containing a comma-separated list as follows: "*system_name:unique_id*".

duObjectDeletionEvent

This event message informs registered applications when an object is deleted from the Viewer's current view.

Inputs: This message is engendered by any object-deletion event occurring from either

- GUI manipulation within the Viewer, or
- Another PMI-based application (such as `obed`).

Processing: When an object deletion is seen by the viewer, it checks to see if that object is the one currently being displayed by the viewer. If it is, the viewer issues a viewerObjectDeletionMessage.

Outputs: The action generated is a `viewerObjectDeletionEvent`. The parameter accompanying the messages is a quoted string containing a comma-separated list as follows: "*system_name:unique_id*".

duViewChangeEvent

This event notifies applications when the user (or another application) changes the view that the Viewer is currently displaying.

Inputs: A change of view causes this message to be generated. Either the user of the program can change the view.

Processing: When the Viewer successfully changes the view it displays, it generates a message containing the new view name. If no applications have registered with the Viewer, no message is generated.

Outputs: The action generated is viewerViewChange. The parameter accompanying the message is a quoted string containing the "*system_name:unique_id*" of the new view being displayed.

duRegisterForEvents

This message is never sent by the viewer to any application. See event description in Table 1-3.

1.2.4 Sample Programs

Sample programs using methods provided by the ViewerApi class are located in the `/opt/SUNWconn/em/src/appapi` directory. For example, the program `viewerapi_eventtester.cc` gives an example of how to use event handling with the Viewer.

For an example of the application instance object of a running viewer refer to `subsystemId="EM-MIS"/emApplicationID=8`. The example program, `viewerapi_driver`, uses the integer `8` to create a singular target ID. Requests can be sent to this target ID.

For more information on actions, refer to the GDMO and ASN.1 documents, `em_apps_msg.gdmo` and `em_apps_msg.asn1`.

1.3 Grapher API

A Grapher API is available with Enterprise Manager. A grapher library (`libemgraphapi.a`) is located (by default) in `/opt/SUNWconn/em/lib` and a header file (`emgraphapi.h`) is located (by default) in `/opt/SUNWconn/em/include/grapher`.

Graphs are plotted along a grid surface whose X and Y values increase in a linear fashion (such as time). The X and Y grid array values must increase with the array indices. Each <X,Y> pair define a point on the graph, which has a Z value. The Z value is the variable value that is plotted for the dataset.

Each graph that is created has a graph name and the name of the client application associated with it. A dataset is registered only once with the Grapher when dynamic plotting is used. If you do not intend to use the Grapher for dynamic plotting, you can register the same dataset more than once. For example, you might want to do this to display graphs with the same data but with different colors or plot methods at the same time. You can define a blank spot in a graph, called a “hole,” by specifying a large double value as the value for that point. This value must be set when the dataset is created.

A sample program using the Grapher API is included with the Solstice EM product. The default directory for the sample program is `/opt/SUNWconn/em/src/grapher`. This directory includes a `README` file that discusses the sample program as well as the Grapher API.

The Grapher API section includes the following classes:

- *EMdataset Class*
- *EMdynamicDataset Class*
- *EMStaticDataset Class*
- *Err Class*

1.3.1 `EMdataset` *Class*

`EMdataset:` A common base class for both `EMStaticDataset` and `EMdynamicDataset`.

1.3.2 `EMdynamicDataset` *Class*

`EMdynamicDataset:` A class for storing attributes of a dynamic dataset such as number of points to be displayed at a time. A dynamic dataset is capable of being updated with new values. The number of grid points along the Y axis (the graph depth) are fixed by the first call for the dataset. This value cannot be changed after it is set and it is necessary to provide the same number of points along the Y axis for subsequent update calls. If you do not have values for certain points along the Y axis, you can specify “noval” as the value. “noval” is a large double value that you specify at creation of the dynamic dataset. All such locations are shown as holes in the graph.

1.3.2.1 Constructor

```
EMdynamicDataset::EMdynamicDataset(
    char *name,           //Name of the dataset
    int mode,             //Mode of dynamic display: Absolute,
                          //Cumulative, or Delta
    int max_view_points, //Maximum number of X grid points visible
                          //at a time
    double noval)         //Very large double value used to
                          //represent a hole in the graph
```

This constructor function creates a new dynamic dataset in the Grapher API.

1.3.3 `EMgraph` *Class*

`EMgraph:` This is a class for creating a new graph. The graph can consist of one or more datasets.

1.3.3.1 Constructor

```
EMgraph::EMgraph(
    char *graph_title,     //Title of the graph
    char *client_name)    //Name of the client creating the graph
```

This constructor function creates a new graph in the Grapher.

1.3.3.2 Member Functions

The following are member functions for the `EMgraph` class.

add_dataset

```
EMhandle
EMgraph::add_dataset(
    EMdataset *dataset) //A dataset
```

This function adds datasets to a graph.

add_values

```
EMgraph::add_values(
    EMhandle hnd,      //Handle of the dataset obtained when the
                      // dataset is added to the graph
    int numx,          //Number of X grid points
    int numy,          //Number of Y grid points
    double *xgrid,     //Values of X grid points
    double *ygrid,     //Values of Y grid points
    double *zvalues)   //Data values for numx*numy grid points
```

This function adds new values to a dataset.

draw

```
void EMgraph::draw()
```

This function draws a new graph in the Grapher.

1.3.4 `EMStaticDataset` *Class*

`EMStaticDataset`: A class for storing values to be graphed statically. It consists of grid surface data in an X and Y direction with Z values associated with every <X,Y> pair.

1.3.4.1 Constructor

```
EMStaticDataset::EMStaticDataset(
    char *name          //Name of the dataset
    int numx,           //Number of X grid points
    int numy,           //Number of Y grid points
    double *xgrid,      //Values of X grid points
    double *ygrid,      //Values of Y grid points
    double *zvalues,    //Data values for each of numx*numy grid
points
    double noval)      //Very large double value used to represent
                        //a hole in the graph
```

This constructor function creates a new static dataset in the Grapher.

1.3.5 `Err` Class

`Err`: This is a class to provide error checking capability. Both `EMgraph` and `EMdataset` are inherited from it.

1.3.5.1 Member Functions

The following are member functions for the `Err` class.

GetErrType

```
ErrType Err::GetErrType() const
```

This function returns an error type. Error types are listed in `emgraphapi.h.` The sample program under `/opt/SUNWconn/em/src/grapher` illustrates how to check error conditions.

GetErrStr

```
char* Err::GetErrStr() const
```

This function returns an error string that can be printed or copied.

1.4 Application-to-Application API

The Application-to-Application API uses the following classes to aid communication with the emApplicationInstance object.

- *AppInstComm Class*
- *AppInstObj Class*
- *AppRequest Class*
- *AppTarget Class*

The emApplicationInstance object is created for any application that connects to the MIS. An application can send a message to another EM application through this object using the actions emSendApplicationMessage and emSendApplicationReply.

When an action has been sent to an application instance, the application represented by that particular emApplicationInstance object is notified by either an emApplicationMessage or emApplicationReply events. The emitting of the events to the applications is achieved through the behavior of the emApplicationInstance object.

1.4.1 `AppInstComm` Class

This class is used to establish communication with the platform which communicates with another EM application. All messages are sent and received through an instance of this class.

1.4.1.1 Constructors

```
AppInstComm(Platform &platform)
```

Constructs an instance that maintains a connection to *platform*. The *platform* instance must be in the connected state.

1.4.1.2 Member Functions

The following are member functions for the `AppInstComm` class.

build_target

```
static AppTarget
build_target (
    DU &apptype,
    Array(DU) &userid = Array(DU)(),
    Array(DU) &display = Array(DU)()
);
```

This function builds the target set of applications.

DataFormatter

A DataFormatter function can be defined for post- and pre-processing of message data. The function defined as DataFormatter takes the arguments *data* and *action*, does some processing, and returns new data.

action is the PARAMETER defined in GDMO, while *info* is the string representation of the ASN.1 syntax defined for the PARAMETER.

```
typedef DU (*DataFormatter)(DU &data,DU &action)
```

reply_data_formatter

```
static DataFormatter reply_data_formatter
```

Allows the sender to pre-process the reply data before the sender's callback is executed.

request_data_formatter

```
static DataFormatter request_data_formatter
```

Allows the sender to pre-process the data of the message before sending it to the target application.

indication_data_formatter

```
static DataFormatter indication_data_formatter
```

Allows receiver of a message indication to pre-process the message before the receivers callback is executed.

response_data_formatter

```
static DataFormatter response_data_formatter
```

Allows the receiver of a message to pre-process the message before the reply is sent back to the caller.

send_request

```
static DU send_request(AppInstObj &to, DU &action, DU &info=DU())
```

Sends the `action` with data `info` to the application represented by the `AppInstObj` class.

send_request_unconfirmed

```
static Result
send_request_unconfirmed(
    AppInstObj &to,
    DU &action,
    DU &info=DU()
)
```

Same as `send_request` but there is no reply to the message.

start_send_request

```
static AppRequest
start_send_request(
    AppInstObj &to,
    DU &action,
    DU &info=DU()
)
```

Same as `send_request`, but the response is asynchronous. Use the `AppRequest` member functions `begin()`, `wait()`, and `get_reply_data()` with this call.

send_request for multiple targets

```
static Array(AppRequest)
send_request(
    AppTarget &target,
    DU  &action,
    DU &info=DU()
)

static Array(AppRequest)
start_send_request(
    AppTarget &target,
    DU &action,
    DU &info=DU()
)

static Result
send_request_unconfirmed(
    AppTarget &target,
    DU &action,
    DU &info=DU()
)
```

These member functions are the same as the previous ones, except they are used for sending to multiple target applications. This is achieved by using the `AppTarget` class instead of the `AppInstObj` class.

set_indication_handler

```
static AppEventHandler
set_indication_handler(
    AppEventHandler callback,
    const DU &pmt
)
```

This function calls the handler defined by *callback* when an indication of type *pmt* is received. *pmt* must be defined as one of the GDMO PARAMETER templates.

set_default_indication_handler

```
static AppAllEventsHandler
set_default_indication_handler(
    AppAllEventsHandler callback
)
```

This function calls the handler *callback* for any application message that is not handled.

send_response

```
static Result
send_response(
    AppInstObj &whosent,
    int id,
    DU &reply_action,
    DU &reply_data
)
```

This member function is called after a successful return from an indication handler. This function is not called when the sender sends the request in unconfirmed mode.

1.4.2 `AppInstObj` *Class*

This class is used to identify the application to which the predefined messages are sent. One instance of this class represents one application.

This is the class which is used to represent the target application for sending messages. If you want to send messages to multiple applications at once, use the `AppTarget` class.

1.4.2.1 Constructors

```
AppInstObj(int id)
```

Constructs an instance of `AppInstObj`, which represents an EM application. The argument *id* is the naming attribute of the emApplicationInstance object that this instance represents.

```
AppInstObj(DU &fdn)
```

Constructs an instance of `AppInstObj` which represents an EM application. The argument *fdn* is the fully distinguished name of the `emApplicationInstance` object which this instance represents.

```
AppInstObj(Platform&)
```

Constructs an instance of `AppInstObj` which represents the current application.

```
AppInstObj(Image &im)
```

Constructs an instance of `AppInstObj` which represents an EM application. The argument *im* is an `Image` which contains the find of the emApplicationInstance object which this instance represents.

1.4.2.2 Member Functions

get_objname

```
DU get_objname()
```

Returns the fully distinguished name of the application object instance represented by this instance of `AppInstObj`.

get_oi

```
Asn1Value get_oi()
```

Returns the fully distinguished name of the application object instance represented by this instance of `AppInstObj`.

1.4.3 `AppRequest` *Class*

This class represents one message. The message can be synchronous or can be asynchronous waiting to complete.

1.4.3.1 Constructor

```
AppRequest(AppInstObj &to, DU &action,DU &info,int id=0)
```

1.4.3.2 Member Functions

begin

```
Result begin()
Result wait(Timeout api_timeout = API_DEFAULT_TIMEOUT)
```

`begin()` and `wait()` are used in conjunction with asynchronous `send_request functions. begin()` calls the request function and `wait()` blocks until the reply has been received for the message.

get_reply_data

```
DU get_reply_data()
```

After a `wait()` has returned from an asynchronous `send_request`, `get_reply_data()` is used to get the message data.

get_receiver

```
AppInstObj get_receiver()
```

Gets the `AppInstObj` that represents this `AppRequest`. This can be used when multiple replies are received.

get_action

```
DU get_action()
```

Gets the name of the message type which corresponds to this `AppRequest`. The DU returned contains a GDMO PARAMETER template name.

is_complete

```
int is_complete()
```

Returns 1 means the `AppRequest` is completed, a reply has been received. Returns 0 means the reply has not been received. This can be used in conjunction with the asynchronous `send_request` methods.

1.4.4 Actions

`emSendApplicationMessage:` This action is used to send a message to an EM application represented by the emApplicationInstance object the action is sent to.

Syntax:

```
EMSendApplicationMessage ::= SEQUENCE {
messageId        EMMessageID,
messageType      EMMessageType,
message          ANYDEFINEDBY messageType
}
```

messageId EMMessageID (INTEGER): Agreed-upon identifier for messages

messageType (OBJECT IDENTIFIER): This parameter is an OID for a PARAMETER type defined in GDMO. The parameter defines what the ASN.1 syntax is for the specified message type.

emSendApplicationReply: This action allows an EM application to send a reply to the EM application represented by this emApplicationInstance object.

Syntax is the same as for emSendApplicationMessage:

```
emSendApplicationReply ::= SEQUENCE
messageId       EMMessageID,
messageType     EMMessageType,
reply           ANYDEFINEDBY messageType
}
```

The messageType, message, and reply fields for these actions must be agreed upon between applications.

1.4.5 Notifications

emApplicationMessage: This notification is emitted when another application performs an emSendApplicationMessage action on an emApplicationInstance object.

```
EMApplicationMessage ::= SEQUENCE {
sender              EMApplicationOI,
COMPONENTS OF        MSendApplicationMessage
}
```

sender OBJECT INSTANCE: The fdn of the emApplicationInstance of the application which triggered the action.

COMPONENTS OF EMSendApplicationMessage: This contains the messageID, messageType, and message data which was part of the original action.

`emApplicationReply`: This notification is emitted when another application performs an `emSendApplicationReply` action on an emApplicationInstance object.

Syntax:

```
EMApplicationReply ::= SEQUENCE {
sender                   EMApplicationOI,
COMPONENTS OF             EMSendApplicationReply
}
```

Both of these notifications contain the same information from the original action, and also identify the application that sent the action.

1.4.6 An Example

Suppose you have two applications that want to keep track of each other. You have decided the best way to do this is to use an application-to-application "ping" function. This way the applications can ping each other to make sure they are alive. (See Code Example 1-1 on page 1-32 and Code Example 1-2 on page 1-33 for such an application.)

You need two messages for the application: `appHello` and `appAlive.`

The steps you follow are shown in the sections below.

1. **Define the GDMO PARAMETER templates for these messages.**
   The sample below indicates how you can define these templates.

```
MODULE “EM APP PING”

appHello PARAMETER
    CONTEXT  ACTION-INFO;
    WITH SYNTAX  FORUM-ASN1-1.GraphicString30;
    BEHAVIOUR appHelloBehaviour;

REGISTERED AS { 1 2 3 4 5 6 90 };

appHelloBehavior BEHAVIOUR

DEFINED AS
    !Ping hello message for app to app communication!;

appReply PARAMETER
    CONTEXT  ACTION-INFO;
    WITH SYNTAX FORUM-ASN1-1.GraphicString30;
    BEHAVIOUR appReplyBehaviour;

REGISTERED AS { 1 2 3 4 5 6 91 };

appReplyBehaviour BEHAVIOUR

DEFINED AS
    !Ping reply message for app to app communication!;

END
```

2. **Compile the GDMO into the MIS.**

```
% em_gdmo hostname app_ping.gdmo
```

3. **Write a driver and a listener application.**
   The driver sends the appHello message with a GraphicString30 string as data. The listener replies with an appReply message containing a GraphicString30 string as data. Make sure to link with `libappapi.so`.

4. **Start the listener first.**
   The listener prints out its application Id. Run the driver with the id of the listener as `arg1`.

   This is one way to identify what application you want to communicate with. You can also dynamically send messages to applications based on their names and what user started them. See the sample programs for more information.

Code Example 1-1 Listener Application

```
// Listener Application for App2App API

#include <hi.hh>
#include “app_comm.hh”

Platform plat;
int handler(DU,DU&,DU&);

main(int argc, char **argv)
{
        plat = Platform(duEM);
        char *host = “localhost”;
        printf(“Connecting to %s ... “,host);
        if (!plat.connect(host, “sample_app2app”)) {
                printf(“Connect failed\n”)

”, plat.get.error.string());
                exit(0);
        }
        printf(“Connected. \n”);
        AppInstComm app(plat);
        AppInstObj::appObj(plat);
        printf(“ID %s\n”,appObj.get_objname().chp());
        app.set_indication_handler( handler,”appHello”);
        while (1)
                dispatch_recursive(TRUE);
}

int
AppEventHandler(
        DU input_info,
        DU &reply_action,
        DU &reply_info,
        AppInstObj sender
)
{
        printf(“Message Received %s\n”,input_info.chp());
        reply_action = “appReply”;
        reply_info = “\”I am alive\””;
        return 1;
}
```

Code Example 1-2 Driver Program for Listener Application

```
//Driver Program for App2App API

#include <hi.hh>
#include “app_comm.hh”

Platform plat;

main(int argc, char **argv)
{
    if (argc != 2) {
                printf(“Usage: <listener_id>\n”);
                exit(1);
        }
        plat = Platform(duEM);
        char *host = “localhost”;
        printf(“Connecting to %s ... “,host);
        if (!plat.connect(host, “sample_app2app”)) {
                printf(“Connect failed\n”);
                printf(“Reason:%ln”, plat.get.error.string());
                exit(0);
        }
        printf(“Connected. \n”);
        AppInstComm app(plat);
        char    action[500];
        char    info[500];
        strcpy(action , “appHello”);
        strcpy(info , “\”Hello there\””);
        AppInstObj to(atoi(argv[1]));
        DU reply;
        reply = app.send_request(to,DU(action),DU(info));
        printf(“Reply is %s\n”,reply.chp());
}
```

1.4.7 `AppTarget` *Class*

This class is used for sending messages to a set of applications based on the application name and user ids. An instance of this class can represent a group of applications or a single application.

1.4.7.1 Constructor

```
AppTarget(
    int id,
    DU &apptype,
    Array(DU) &userid = Array(DU)(),
    Array(DU) &display = Array(DU)()
);
```

Constructs an `AppTarget` instance that represents a set of applications based on the *apptype* and *userid*. The *apptype* is the name of the application set in the `Platform::connect()` method. The *userid* is an array of the UNIX logins of the users who connect to the platform.

1.4.7.2 Member Functions

The following are member functions for the `AppTarget` class.

num_objs

```
int num_objs()
```

Returns the number of applications represented in the `AppTarget` instance.

first_obj

```
AppInstObj first_obj()
```

Returns the first `AppInstObj` in the `AppTarget` instance. The `AppInstObj` can then be used in a `AppInstComm::send_request()` method.

next_obj

```
AppInstObj next_obj()
```

Returns the next AppInstObj in the AppTarget instance. The AppInstObj can then be used in a `AppInstComm::send_request()` method.

Common API Classes 2

This chapter describes classes and their member functions that can be used by all of the Application Programming Interface (API) libraries.

2.1 Common API Classes

The following lists the classes and their member functions described in this chapter:

Table 2-1 Common API Classes

Class	**Description**
Address Class	Used to contain an address
Arraydeclare Macro	Used to create a class whose structure is an array
Asn1ParsedValue Class	Represents a parsed Asn1Value that is invalidated
Asn1Tag Class	Defines an ANS.1 tag class and value
Asn1Type Class	Used to implement ASN.1 encoding and decoding

Table 2-1 Common API Classes

Class	**Description**
Asn1Value Class	Defines storage and operations for ANS.1 values
Blockage class	Used to manage blocked callback events
Callback class	Used to post and dispatch callback events
Command Class	Used to define unique commands
Config Class	Used to manage database configuration file defaults
DataUnit Class	Used as a basic storage unit for data
Dictionary Class	Provides facilities to classes created by the Dictionarydeclare macro
HashImpl Class	Used to implement a dynamically growing hash table
Hashdeclare Macro	Used to create a hash table class
Hdict Class	Provides facilities to classes created by the Hdictdeclare macro
Oid Class	Defines a container for an object identifier

2.2 *Class Categories*

There are five general categories of classes that can be used with the Application Programming Interface:

- Address Classes
- ASN.1 Classes
- Scheduling and Callback Handling Classes
- Array and Hashing Class
- Dictionary Macro Classes

Table 2-2 Class Categories

Category	**Class/Macro**
Address	*Address Class*
Address	*Command Class*
Address	*Config Class*
Asn1	*Asn1ParsedValue Class*

Table 2-2 Class Categories

Category	**Class/Macro**
Asn1	*Asn1Tag Class*
Asn1	*Asn1Type Class*
Asn1	*Asn1Value Class*
Asn1	*DataUnit Class*
Asn1	*Oid Class*
Scheduling/Callback	*Blockage class*
Scheduling/Callback	*Callback class*
Scheduling/Callback	*Timer Class*
Arrays/Hashing	*Hashdeclare Macro*
Arrays/Hashing	*HashImpl Class*
Dictionary	*Dictionary Class*
Dictionary	*Hdict Class*

2.3 *Variables Types*

The basic types of variables are declared in the `/opt/SUNWconn/em/include/pmi/basic.hh` file.

Table 2-3 Basic Variable Types

Variable	**Description**
#define TRUE (1) #define FALSE (0)	
typedef signed char I8;	Signed 8 bit character
typedef unsigned char U8;	Unsigned 8 bit character
typedef short I16;	Signed 16 Bit character
typedef unsigned short U16;	Unsigned 16 Bit character
typedef long I32;	Signed 32 Bit character
typedef unsigned long U32;	Unsigned 32 Bit character
typedef U8 Boolean;	General type used to distinguish true from false by 0 or nonzero.

Table 2-3 Basic Variable Types

typedef unsigned char Octet;	Unsigned char
typedef void *Ptr;	Void pointer
typedef enum Result {OK, NOT_OK };	Enumerated type with 2 values, used as return code from many functions. The
typedef U32 MTime;	Unsigned 32 bit quantity to hold time in Milliseconds
const MTime INFINITY = 0xffffffff;	Constant value

2.4 Class Descriptions

The following lists the classes that can be used by the API libraries:

Table 2-4 Common API Classes

Class	**Description**
Address Class	Used to contain an address
Asn1ParsedValue Class	Represents a parsed Asn1Value that is invalidated
Asn1Tag Class	Defines an ANS.1 tag class and value
Asn1Type Class	Used to implement ASN.1 encoding and decoding
Asn1Value Class	Defines storage and operations for ANS.1 values
Blockage class	Used to manage blocked callback events
Callback class	Used to post and dispatch callback events
Command Class	Used to define unique commands
Config Class	Used to manage database configuration file defaults
DataUnit Class	Used as a basic storage unit for data
Dictionary Class	Provides facilities to classes created by the Dictionarydeclare macro
HashImpl Class	Used to implement a dynamically growing hash table
Hdict Class	Provides facilities to classes created by the Hdictdeclare macro
Oid Class	Defines a container for an object identifier

2.4.1 Address Class

Inheritance: `class Address`

`#include <pmi/address.hh>`

This class defines a structure used to contain an address. An instance of `Address` contains an address class, an address tag, and an address value. The address tag is used with the value to form an address. The `Address` class distinguishes between members of the enum `AddressClass`:

Table 2-5 AddressClass Data Members

Address Variables	**Description**
AC_DEFAULT	Route by object instance in request
AC_APP	Application address
AC_DIR_SERVICE	A directory service for resolution
AC_PRIMITIVE	Protocol driver address

The address tag can be one of the values shown in Table 2-6. Refer to Section 2.4.1.5, “AddressTag Variable,” on page 2-7 for more information.

Table 2-6 AddressTag Data Members

Address Tag Variables	**Value**	**Description**
AT_PRIM_OAM	0	OAM MIS module
AT_PRIM_EMM	1	EMM MIS module
AT_PRIM_CMIP_PRES_ADDR	2	CMIP address
AT_PRIM_SNMP_ADDR	3	SNMP address
AT_PRIM_AET_ADDR	4	ASN.1 Address Entity Title
AT_PRIM_MPA_ADDR	5	MPA address
AT_PRIM_AGENT_DN	6	ASN.1 FDN
AT_PRIM_RPC_ADDR	7	RPC address
AT_PRIM_CMIP_CONFIG	8	String consisting of: {PSEL, SSEL, TSEL, NSAPS}

Table 2-7 lists the well-known address types for directory services.

Table 2-7 Directory Services

Directory Service	**Value**
`AT_DS_OBJ_INST`	0
`AT_DS_APP_ENT_TITLE`	1
`AT_DS_DOMAIN_NAME`	2

Table 2-8 lists the `Address` class public variables.

Table 2-8 Address Public Variables

Variable Type	**Parameter**	**Description**
AddressClass	`aclass;`	The address class
AddressTag	`atag;`	The address tag
DataUnit	`aval;`	ASN.1-encoded address value

2.4.1.1 Constructor

```
Address::Address()
```

2.4.1.2 Operator

```
const Address &operator = (const Address &<addr>)
```

The operator above creates an instance of `Address` having the same values as the argument `Address`.

```
int operator == (const Address &<a>)
const
```

This operator returns true if the two Addresses have the same class, tag, and value.

```
int operator != (const Address &<a>) const
```

This operator returns true unless the two `Addresses` have the same class, tag, and value.

2.4.1.3 *Member Functions of Address*

This section describes the member functions of the `Address` class.

print

```
void print(FILE *<fp>) const

void print(Debug &<deb> = misc_stdout) const
```

These function calls append a record of the Address's class, tag, and value to the file <*fp*> or to the debug stream <*deb*>.

2.4.1.4 `AddressClass` *Variable*

The `AddressClass` variable is declared in the `/opt/SUNWconn/em/include/pmi/address.hh` file as follows:

```
enum AddressClass
      AC_DEFAULT,     //route by object instance in request
      AC_APP,         //an application
      AC_DIR_SERVICE, //a directory service
      AC_PRIMITIVE   //a protocol driver
} ;
```

2.4.1.5 `AddressTag` *Variable*

The `AddressTag` variable, as shown in Table 2-6 and Table 2-7, is declared in the `/opt/SUNWconn/em/include/pmi/address.hh` file.

2.4.2 Arraydeclare *Macro*

Declared in: `array.hh`

```
#define Array(T) name2(T, Array)
```

```
#define Arraydeclare(T)\
```

The `Arraydeclare` macro declares a class whose name is formed from "Array" followed by its argument. This permits creation of a class whose structure is an array with each element an instance of the class named in its argument. Because an Array object thus created is confined to the scope in which the macro is used, it deletes itself when the scope is exited. This obviates the need for "array delete" statements before return, since arrays are automatically deleted at return.

The definition for the `overloaded =` operator assures that only one Array can be in charge of a piece of memory at a time. It does this by swiping the array pointer from the other Array. If you want to have multiple people referencing the same array, pass a reference to the Array object, or sneak the pointer out of the Array object. However, do not put the pointer into some other Array object, because that would cause a double delete. The array's size is defined as a "pseudo const" — although the class itself cheats on the `const`-ness, it expects the user not to.

Because the subscripting operator is defined in-line, there should be no extra overhead in using this class over doing the subscripting directly. If you want the subscripting operator to do extra checking, define `SUBSCRIPTCHECK` appropriately.

```
#define SUBSCRIPTCHECK assert(subscript < size)
```

This macro provides validation of the subscripts used in indexing an array, but only while operating in debug mode.

The following are array declarations:

```
Arraydeclare(Boolean);
Arraydeclare(I8);
Arraydeclare(U8);
Arraydeclare(I16);
Arraydeclare(U16);
Arraydeclare(I32);
Arraydeclare(U32);
Arraydeclare(Ptr);
Arraydeclare(Result);
Arraydeclare(char);
```

2.4.3 Asn1ParsedValue *Class*

Inheritance: classAsn1ParsedValue

#include <pmi/asn1_type.hh>

Data Members: No public data members declared in this class

An Asn1ParsedValue represents a parsed Asn1Value that has not yet been validated against a type. This form is used to hold a parsed value until its type has been completely determined. The Asn1ParsedValue is represented internally by an Asn1Value of the type ASN-PARSED-VALUE.

Classes used in various aspects of the implementation of ASN.1 encoding and decoding rely on the ISO specifications of Abstract Syntax Notation One. For details, please consult the sources cited in the standards documents appendix to the Solstice EM *Overview* document. Table 2-9 lists the Asn1ParsedValue public functions.

Table 2-9 Asn1ParsedValue Public Functions

Function Name	**Descriptions**
= ! void*	Operator overloading
get_real_val get_parsed_val format_value	Extract the value in various formats

2.4.3.1 Constructors

```
Asn1ParsedValue()

Asn1ParsedValue(const Asn1Value &<pv>)

Asn1ParsedValue(const Asn1ParsedValue &<pv>)
```

These are constructors for the `Ans1ParsedValue` class

2.4.3.2 Operator Overloading for `Asn1ParsedValue`

```
const Asn1ParsedValue &operator = (const Asn1ParsedValue &<pv>)

operator void *() const
```

These are the operators for the `Ans1ParsedValue` class..

```
int operator !() const
```

This operator is provided so that you can say “`if (!Asn1ParsedValue)`…”

2.4.3.3 Member Functions of `Asn1ParsedValue`

This section describes the member functions of the `Asn1ParsedValue` class.

format_value

```
Result format_value(char *&<buf>,
    U32 &<buf_len>,
    U32 <indent>,
    Asn1Tagging <tagging>) const
```

This function writes into the buffer <*buf*> a string representation of the `Asn1ParsedValue`. The representation must have a length no greater than <*buf_len*> characters, including <*indent*> leading blanks. This function returns OK if the value can be extracted and fits in buffer length provided.

get_parsed_val

```
Asn1Value get_parsed_val() const
```

This function returns the encoded parsed value as an `Asn1Value`.

get_real_val

```
Asn1Value get_real_val(const Asn1Type &<type>) const
```

This function, given an `Asn1Type`, returns an `Asn1Value` representation of the value.

2.4.4 `Asn1Tag` *Class*

Inheritance: `class Asn1Tag`

```
#include <pmi/asn1_val.hh>
#include <pmi/basic.hh>
```

Asn1Tag is a class that defines an ASN.1 tag class and an ASN.1 value for a tag. An ASN.1 tag value is defined as a U32 value. The enumeration Asn1TagClass (see Section 2.4.16.11, “Asn1TagClass,” on page 2-79) has the following possible values:

```
CLASS_UNIV
CLASS_APPL
CLASS_CONT
CLASS_PRIV
```

Classes used in various aspects of the implementation of ASN.1 encoding and decoding rely on the ISO specifications of Abstract Syntax Notation One. For details, please consult the sources cited in the standards document. Table 2-10 lists the Asn1Tag class public variables.

Table 2-10 Asn1Tag Public Variables

Variable Type	**Parameter**	**Description**
Asn1TagClass	`tclass;`	The tag class
Asn1TagValue	`value;`	The value of the tag within the class

Table 2-11 lists the Asn1Tag public functions.

Table 2-11 Asn1Tag Public Functions

Function Name	**Descriptions**
== !=	Operator overloading
size	Report the tag’s size

2.4.4.1 *Constructors*

```
Asn1Tag()
```

These are the constructors for the `Asn1Tag` class. The following constructor creates an instance of an `Asn1Tag` with a tag class of `CLASS_UNIV` and a tag value of 0.

```
Asn1Tag(Ans1TagClass <cl>,Ans1TagValue <val>)
```

This constructor creates an `Asn1Tag` with a tag class specified by <*cl*> and a tag value specified by <*val*>.

2.4.4.2 *Operator Overloading for* `Asn1Tag`

This section describes the operators for the `Asn1Tag` class.

```
int operator == (const Asn1Tag &Tag) const
```

This operator compares the operands' tags and returns nonzero if the two are equal, zero otherwise.

```
int operator != (const Asn1Tag &Tag) const
```

This operator compares the operands' tags and returns nonzero if the two are different, zero otherwise.

2.4.4.3 *Member Functions of* `Asn1Tag`

This sections describes the member function of the `Asn1Tag` class.

size

```
U32 size(Asn1Encoding <enc>)
```

This function call returns a U32 value containing the number of octets in the tag.

2.4.5 `Asn1Type` *Class*

Inheritance: `class Asn1Type`

`#include <pmi/asn1_type.hh>`

Classes used in various aspects of the implementation of ASN.1 encoding and decoding rely on the ISO specifications of Abstract Syntax Notation One. For details, please consult the sources cited in the standards document. The `Asn1Module`, `ATData`, and `Asn1ParsedValue` belong to the friend class.

Table 2-12 lists `Asn1Type` public functions.

Table 2-12 Asn1Type Public Functions

Function Name	**Descriptions**
= void*() !()	Operator overloading
base_kind base_type kind needs_explicit	Report kind and type
format_type	Conversion
add_tags format_value parse_value remove_tags validate validate_tag	Construct an Asn1Value from a string
cmp equivalent	Compare two values
determine_real_val	Return encoded from parsed form

Table 2-12 Asn1Type Public Functions

Function Name	**Descriptions**
find_component find_subcomponent	Find components
set_add_members set_intersects_with set_is_subset set_remove_dup_members set_remove_members	Set operations on Asn1Values (of the type identified by this Asn1Type)
lookup_type	Retrieve type information
register_any_handler unregister_any_handler	Provide user-defined function for the key-to-value lookup

2.4.5.1 Constructors

```
Asn1Type(Asn1Kind <k>)
```

Use this constructor to construct `BOOLEAN`, `OCTET_STRING`, `NULL`, `OBJECT_IDENTIFIER`, and `REAL` entities. Asn1Kind has numeric definitions of enumerated datatypes.

```
enum Asn1Kind {
  AK_NONE, AK_BOOLEAN, AK_INTEGER, AK_BIT_STRING, AK_OCTET_STRING,
  AK_NULL, AK_SEQUENCE, AK_SEQUENCE_OF, AK_SET, AK_SET_OF,
  AK_CHOICE, AK_SELECTION, AK_TAGGED, AK_ANY, AK_OBJECT_IDENTIFIER,
  AK_ENUMERATED, AK_REAL, AK_SUBTYPE, AK_DEFINED_TYPE
};
```

```
Asn1Type(const Asn1Kind &<at>)
Asn1Type(const Asn1Type &<at>);
Asn1Type(const Asn1Type &<av>);
```

Use one of the preceding constructors to initialized from an `Asn1Value` representation of the `Asn1Type`.

```
Asn1Type(const DataUnit &<module_name>,char *)
```

This constructor is initialized from the canonical text representation of the `Asn1Type`.

2.4.5.2 Destructor

```
~Asn1Type()
```

2.4.5.3 Operator Overloading for `Asn1Type`

This sections describes operator overloading for `Asn1Type`.

```
Asn1Type &operator = (const Asn1Type &<at>)
```

This operator tests the `Asn1Type` to see whether it is initialized.

```
operator void *() const
```

This is an operator for the `Ans1ParsedValue` class.

```
operator Asn1Value() const
```

This operator converts the `Asn1Type` to its `Asn1Value` representation.

```
int operator !() const
```

This operator is provided so that you can say “`if (!Asn1ParsedValue)`…”

2.4.5.4 *Member Functions of* `Asn1Type`

This sections describes the member functions of the `Asn1Type` class.

add_tags

```
Asn1Value add_tags(const Asn1Value &<av>) const
```

This function call retags or adds tags to a value to make it conform to the type.

base_kind

```
Asn1Kind base_kind() const
```

This function call returns the underlying “kind” of the type, recursing through `TAGGED`, `SELECTION`, `DEFINED_TYPE`, and `SUBTYPE`.

base_type

```
Asn1Type base_type() const
```

This function call returns type without respect to tagging or selection.

cmp

```
int cmp(const Asn1Value &<av1>, const Asn1Value &<av2>) const;
```

This function call determines the ordering of two `Asn1Values`. Returns a negative, 0 or positive value, as *<av1>* is less than, equal to, or greater than *<av2>*.

determine

```
Asn1Value determine_real_val(const Asn1Value &<parsed>) const;
```

This function call takes an Asn1Value in parsed form and returns it in encoded form.

equivalent

```
Boolean equivalent(const Asn1Value &<av1>,const Asn1Value &<av2>)
const;
```

This function call determines whether two Asn1Values are equivalent this assumes both values have been validated against the type.

find_component

```
Asn1Value find_component(const DataUnit &<field_name>,
   const Asn1Value &<val>,
   Asn1Type &<comp_type>) const;
```

This function call finds the named component of the Asn1Type in the given Asn1Value. A component is a named field for SEQUENCE, SET, CHOICE, or a number for SEQUENCE OF, SET OF (0 is the count of the number of elements). Returns the type of the component in <*comp_type*>.

find_subcomponent

```
Asn1Value find_subcomponent(const DataUnit &<field_name>,

    const Asn1Value &<val>,

    Asn1Type &<comp_type>) const;
```

This function call finds the named subcomponent of the Asn1Type in the given Asn1Value. A subcomponent is a list of component names separated by a period.

format_type

```
Result format_type(char *&<buf>,
    U32 &<buf_len>,
    U32 indent = 0,
    Asn1Tagging <tagging> = TAG_EXPLICIT) const;
```

This function call converts the Asn1Type to its canonical text representation.

```
Result format_type(const Asn1Value &<av>,

    char *&<buf>,

    U32 &<buf_len>,

    U32 indent = 0) const;
```

This function call converts an Asn1Value to its canonical text representation

kind

```
Asn1Kind kind() const
```

This function call returns the “kind” of the type, i.e, BOOLEAN, INTEGER, SEQUENCE, DEFINED_VALUE, etc.

lookup_type

```
static Asn1Type lookup_type(const DataUnit &<any_name>,

     const DataUnit &<defined_by_name>,

     const Asn1Value &<any_value>)
```

This function call returns the Asn1Type associated with a particular key value from an ANY DEFINED BY clause.

In the function definition, above:

<*any_name*> is the name of the field of the sequence or set which is the ANY. <*defined_by_name*> is the name of the field storing the key. <*any_value*> is the value of the key.

An application can use this to discover the syntax of an attributes, action, notification, parameter, and so on. For example, consider the type:

```
SEQUENCE {
key OBJECT IDENTIFIER
value ANY DEFINED BY KEY }
```

The expression

```
Asn1Value::lookup_type("value", "key", myOID)
```

would return the `Asn1Type` associated with this particular combination of the three arguments.

The method by which `lookup_type` identifies a particular key-and-value combination can be provided explicitly by the static method `register_any_handler` (below). After `register_any_handler` has been invoked, `lookup_type` first searches to find a function registered with an exact match in the <*any_name*> and <*defined_by_name*> fields. Then it tries to find the function registered with a matching <*any_name*> but an empty <*defined_by_name*> field. Finally it tries to find a function with both names empty. If it can't find a function, or if the function can't locate the key, it returns a null `Asn1Type` as the result of `lookup_type`.

needs_explicit

```
Boolean needs_explicit() const
```

This function call returns whether type requires explicit tagging.

parse_value

```
Ans1Value parse_value(const DataUnit &<module_name>,char *) const;
```

This function call constructs an `Asn1Value` from a string.

register_any_handler

This function call specifies a key-to-value lookup function to be used by `lookup_type` (discussed above) for a particular combination of <*any_name*> and <*defined_by_name*>. Usually `register_any_handler` is invoked automatically by the PMI, but an application can invoke it explicitly.

After `register_any_handler` has been invoked, `lookup_type` first searches to find a function registered with an exact match in the <*any_name*> and <*defined_by_name*> fields. Then it tries to find the function registered with a matching <*any_name*> but an empty <*defined_by_name*> field. Finally it tries to find a function with both names empty. If it can't find a function, or if the function can't locate the key, it returns a null `Asn1Type` as the result of `lookup_type`. See also Asn1Type::`unregister_any_handler` and Asn1Type::`lookup_type`.

remove_tags

```
Asn1Value remove_tags(const Asn1Value &<av>) const
```

This function call removes explicit tags from a value to get to the underlying base value.

set_add_members

```
Result set_add_members(const Asn1Value &<of>,Asn1Value &<to>) const;
```

This function call's arguments <*of*> and <*to*> are `Asn1Values`. Both of them are of the type identified by this `Asn1Type`. The unique members of <*of*> are inserted into <*to*>.

Returns `OK` if the operation completed successfully, and `NOT_OK` otherwise (for example, if the `Asn1Values` were not both of the appropriate type).

set_intersects

```
Boolean set_intersects_with(const Asn1Value &<left>,Asn1Value &<right>)
const;
```

This function call's arguments <*left*> and <*right*> are Asn1Values. Both of them are of the type identified by this Asn1Type.

Returns OK if <*left*> is a subset of <*right*>; that is, if all the members of <*left*> are also members of <*right*>.

set_is_subset

```
Boolean set_is_subset(const Asn1Value &<left>,

                      const Asn1Value &<right>),

                      Boolean <mutual> = FALSE) const;
```

This function call's arguments <*left*> and <*right*> are Asn1Values. Both of them are of the type identified by this Asn1Type.

Returns OK if <*left*> is a subset of <*right*>; that is, if all the members of <*left*> are also members of <*right*>.

set_remove_dup_members

```
Result set_remove_dup_members(Asn1Value &<of>) const;
```

This function call's argument <*of*> is an Asn1Values of the type identified by this Asn1Type. The method removes any duplicate members from <*of*>.

Returns OK if the operation completed successfully, and NOT_OK otherwise (for example, if the Asn1Values were not both of the appropriate type).

set_remove_members

```
Result set_remove_members(const Asn1Value &<of>, Asn1Value &<from>)
const;
```

This function call’s arguments <*of*> and <*from*> are `Asn1Values`. Both of them are of the type identified by this `Asn1Type`. The method removes from <*from*> any members that are also present in <*of*>.

Returns `OK` if the operation completed successfully, and `NOT_OK` otherwise (for example, if the `Asn1Values` were not both of the appropriate type).

unregister_any_handler

```
static result unregister_any_handler(const DataUnit &<any_name>,

     const DataUnit &<defined_by_name>,

     AnyHandler &<handler>)
```

This function call undoes the effect of registering an `AnyHandler` function to work with `lookup_type`.

validate

```
Result validate(const Asn1Value &<av>, Boolean <ignore_tag> = FALSE)
const;
```

This function call determines whether the specified `Asn1Value` is correctly formatted for the type.

validate_tag

```
Result validate_tag(const Asn1Value &<av>) const;
```

This function call determines whether the specified `Asn1Value` has the correct tag for the type.

2.4.6 `Asn1Value` *Class*

Inheritance: `class Asn1Value`

`#include <pmi/asn1_val.hh>`

The `Asn1Value` class defines storage and operations for ASN.1 values. The class is capable of supporting multiple encoding schemes but presently only supports Basic Encoding Rules (BER) encoding and encoding in a native machine format. The `Asn1Value` class uses the `DataUnit` class to store encoded values. It can also use other classes to store special types of data. To store unencoded data, the `Asn1Value` class makes use of the protected class `AVData` (declared as a class but not otherwise specified in `/opt/SUNWconn/em/include/pmi/asn1_val.hh`).

The `new` and `delete` operators should not be used with the `Asn1Value` constructor and destructor functions because the class manages and allocates memory for the values it stores.

2.4.6.1 *Assignment and Data Sharing*

The = operator is overloaded to permit assignment of one instance of `Asn1Value` to another, so that both refer to the same data.

2.4.6.2 *Type Conversion*

The `Asn1Value` class facilitates type conversion by overloading the `void *()` and `DataUnit` operators.

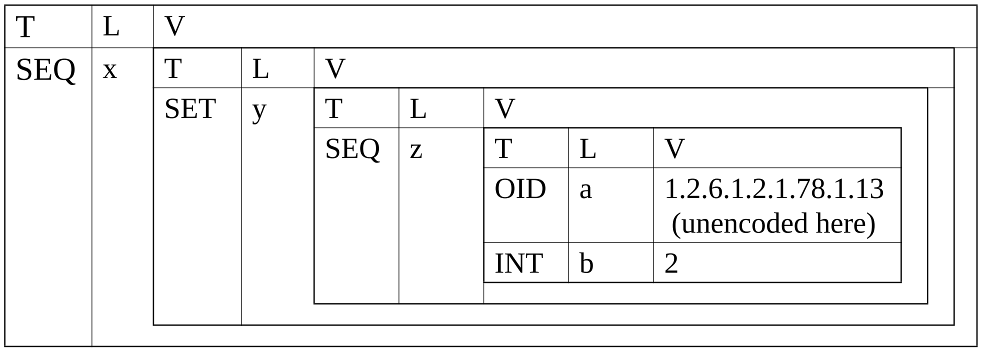
2.4.6.3 *Encoding*

The class defines a set of encoding functions. Each allocates an `AVData` instance to store a value in native machine encoding. If the `Asn1Value` previously contained a value, the memory for it is freed prior to allocating memory for the new value. These functions normally return OK, unless memory cannot be allocated for the new value, or under certain conditions noted in the descriptions that follow.

In general, the functions do not actually perform the encoding. In most cases the `Asn1Value` class handles the encoding automatically the first time a function requests the data in an encoded form.

2.4.6.4 *Encoding of a Distinguished Name*

A distinguished name (DN) is represented by a constructed ASN.1 value. Consider a DN containing a single RDN that has an id=1.2.6.1.2.1.42.1.13 and a value=2. It would have the following format:



A code fragment to construct that distinguished name might be as follows:

```
Octet*id = “1.3.6.1.2.1.42.1.13”;

I32value = 2;

Asn1Value dn_ex1;
Asn1Value rdn_ex1;
Asn1Value ava_ex1;
Asn1Value id_ex1;
Asn1Value val_ex1;
Result status_ex1 = OK;

if (
dn_ex1.start_construct(TAG_SEQ) != OK ||// init the RDN
rdn_ex1.start_construct(TAG_SET) != OK ||// init the RDN
ava_ex1.start_construct(TAG_SEQ) != OK || // init the AVA
id_ex1.encode_oidstr((Octet *)id, TAG_OID) != OK || // encode the OID
val_ex1.encode_int(value, TAG_INT) != OK || // encode the integer
ava_ex1.add_component(id_ex1)) != OK || // create the AVA from
ava_ex1.add_component(val_ex1) != OK || // ...OID and integer
rdn_ex1.add_component(ava_ex1)!= OK // complete the RDN
dn_ex1.add_component(rdn_ex1) != OK // complete the DN
)

status_ex1 = NOT_OK;
```

2.4.6.5 *Decoding Simple and Constructed* Asn1Values

The Asn1Value class defines functions for decoding both simple and constructed Asn1Values. If a simple Asn1Value stores the value in the native machine format specified by the function, it is returned directly. If the value is stored in an encoded form, it is decoded, stored in the Asn1Value in the decoded form (it also still exists in the encoded form) and the returns the decoded value.

The functions return NOT_OK if called for an uninitialized Asn1Value, if called for a constructed value, or if called for an Asn1Value that is initialized but contains neither a decoded value or encoded value of the proper type.

Classes used in various aspects of the implementation of ASN.1 encoding and decoding rely on the ISO specifications of Abstract Syntax Notation One. For details, please consult the sources cited in the standards documents appendix to the Solstice EM *Overview* document. Table 2-13 lists Asn1Value functions.

Table 2-13 Asn1Value Functions

Function Name	**Descriptions**
constructed contents_size encoding incl_embedded size tag	Query Asn1Value info
encode_bits encode_boolean encode_enum encode_ext encode_int encode_null encode_octets encode_oid encode_oidstr encode_real encode_unsigned	Encoding Asn1Value simple types

Table 2-13 Asn1Value Functions

Function Name	**Descriptions**
decode_bits decode_boolean decode_enum decode_ext decode_int decode_octets decode_oid decode_real decode_unsigned	Decoding Asn1Value simple types
add_component make_explicit_tagged start_construct	Encoding constructed Asn1Values
delete_component first_component make_explicit_tagged next_component retag tagged_component	Decoding constructed Asn1Values

2.4.6.6 Constructors

```
Asn1Value();
```

The following constructor allocates storage for an uninitialized ASN.1 value. Memory is not allocated for an `AVData` instance or for a `DataUnit` instance. The constructor simply creates an instance of an `Asn1Value` and then sets its `AVData` pointer to 0.

No memory is allocated for the `AVData` instance or for a `DataUnit` in the constructor. If the encoding specified by the <*enc*> parameter matches the encoding specified for `Asn1Value` specified by the <*av*> parameter, the constructor instantiates an `Asn1Value` that points to the `AVData` contained in

<*av*> (that is, it shares data with <*av*>). The reference count for the `AVData` instance pointed to by <*av*> is incremented. The default encoding for this constructor is BER; currently only BER is supported.

```
Asn1Value(Asn1Value &<av>, Asn1Encoding <enc> = ENC_BER);
```

The `new` operator should not be used in conjunction with the above constructor function.

In the following constructor, memory is allocated for an `AVData` instance and the `AVData` instance is pointed at the `DataUnit` referenced by <*du*>. The reference count for the `AVData` instance created is set to one. No memory is allocated for the `DataUnit`. If the `DataUnit` specified by <*du*> is not valid, the pointer to the `AVData` instance is set to 0 and the memory for the `AVData` instance is deallocated.

```
Asn1Value(const DataUnit &<du>, Asn1Encoding <enc> = ENC_BER);
```

The `new` operator should not be used with the above constructor function.

In the following constructor, memory is allocated for an `AVData` instance, but the `DataUnit` is shared with the `DataUnit` specified by the <*du*> parameter. The tag for the `Asn1Value` is set to the value specified by the <*tag*> parameter.

An `Asn1Value` can be a constructed type or a simple type. The <*constr*> parameter specifies whether the value specified by <*du*> is a constructed type. The < *type*> attribute of the newly instantiated `Asn1Value` is set to the value specified by <*constr*>.

```
Asn1Value(const Asn1Tag &<tag>,

    BOOLEAN <const>,

    const DataUnit &<du>,

    Asn1Encoding <enc> = ENC_BER);
```

The `new` operator should not be used with the above constructor function.

2.4.6.7 Destructor

```
~Asn1Value();
```

The destructor decrements the reference count for the `AVData` instance pointed at by this `Asn1Value`. If the reference count goes to zero, the memory for the `AVData` instance is deallocated.

2.4.6.8 Operator Overloading for `Asn1Value`

The = operator is overloaded so that one `Asn1Value` can share data with another `Asn1Value`. In the returned `Asn1Value`, the `AVData` pointer points to the same object as the `AVData` pointer in <*av*>. At the same time, assignment increases the reference count for the `AVData` unit that <*av*> points to.

```
Asn1Value &operator = (Asn1Value &<av>);
```

If the instance of the `Asn1Value` that this function is operating was pointed to another `AVData` instance, the reference count for the instance is decremented; if the reference count goes to zero, the memory for instance is deallocated. The '=' operator returns a reference to an `Asn1Value`.

```
operator void *() const;
```

The operator `void *()` returns a void pointer to the `AVData` instance pointed to by the `Asn1Value`. This function performs a type conversion from an `Asn1Value` to a `void *` and can be used for implicit type conversions.

```
operator DataUnit() const;
```

The operator `DataUnit` performs a type conversion from an `Asn1Value` to a `DataUnit`. If the `Asn1Value` is uninitialized, it returns a zero length `DataUnit`. This function can be used for implicit type conversions.

```
operator !() const
```

This allows you to say “`if(!Asn1Value...)`”

2.4.6.9 *Member Functions of* `Asn1Value`

This section describes the member functions of the `Asn1Value` class.

add_component

```
Result add_component(Asn1Value &<comp>)
```

This function call adds a component to this constructed `Asn1Value`. Returns OK if the operation completes normally, but NOT_OK if it encounters any of the following conditions:

- The `Asn1Value` has not been initialized;
- The `Asn1Value` is not a constructed type;
- The `Asn1Value` specified by component has not been initialized;
- The encoding of `Asn1Value` specified by component does not match the encoding of the `Asn1Value` to which the component is being added; or
- Memory cannot be allocated to add the new component.

The size of the constructed value is update to reflect the size of the component added.

constructed

```
Boolean constructed()
```

This function call returns a boolean value indicating if the `Asn1Value` is a constructed type. (An `Asn1Value` can either be a constructed type or a simple type.)

contents_size

```
Result contents_size(U32 &<sz>,Boolean <inc_embed> = FALSE)
```

This function call sets the value of the <*sz*> parameter to the total size of a constructed `Asn1Value`. This function decodes each component of a constructed `Asn1Value` and returns the sum of the size of each `DataUnit` in the constructed value. If the `Asn1Value` does not decode properly, this function returns `NOT_OK`; otherwise it returns `OK`.

decode_bits

```
Result decode_bits(DataUnit &<du>,U32 &<len>)
```

This function call decodes an encoded boolean value and stores the result in the `DataUnit` specified by the <*du*> parameter. The length of the bitstring, in bits, is assigned to the parameter specified by <*len*>.

```
Result decode_bits(Octet *<val>,U32 &<len>)
```

This function call decodes an encoded boolean value and store the result in the Octet string pointed to by the <*val*> parameter. The length of the bit string, in bits, is assigned to the parameter specified by <*len*>.

decode_boolean

```
Result decode_boolean(Boolean &<val>)
```

This function call decodes an encoded boolean value and stores the result in the variable referenced by <*val*>.

decode_enum

```
Result decode_enum(I32 &<val>);
```

This function call decodes an enumerated value and stores the result in the variable referenced by the <*val*> parameter.

decode_ext

```
Result decode_ext(Oid &<oid>,

     I32 &<indirect>,

     DataUnit &<odes>,

     Asn1Value &<encoding>)
```

This function call decodes an ASN1 external value, decomposing it into its components by setting the variables shown in Table 2-14 below.

Table 2-14 `decode_ext` Variable Descriptions

Variable	**Description**
oid	Its (optional) object identifier
indirect	Its (optional) indirect reference
odes	Its (optional) data-value descriptor
encoding	Its choice of the following, indicated by its tag: [0] ANY [1] Implicit octet string [2] Implicit bit string

decode_int

```
Result decode_int(I32 &<val>);
```

This function call decodes an encoded I32 value and stores the result in the variable referenced by <*val*>.

decode_octets

```
Result decode_octets(DataUnit &<du>);
```

This function decodes an encoded octet string.

```
Result decode_octets(Octet *<val>, U32 &<len>);
```

This function calls decode an octet string of length <*len*>, and stores a copy of the decoded string into the Octet string pointed to by <*val*>.

decode_oid

```
Result decode_oid(Oid &<oid>);
```

This function calls decode an OID.

```
Result decode_oid(Octet *<val>, U32 &<len>);
```

This function calls decode an OID (Object IDentifier). The argument can be the address of an OID, or a pointer to an octet string and the string's length. The decoding is formed by calling `decode_octets`. If the `oid` is not constructed, the function returns `NOT_OK`; otherwise it returns the result of `decode_octets`.

decode_real

```
Result decode_real(double &<val>);
```

This function call decodes an encoded real value and stores the result in the variable referenced by <*val*>.

decode_unsigned

```
Result decode_unsigned(U32 &<val>);
```

This function call decodes an encoded U32 value and stores the result in the variable referenced by <*val*>.

delete_component

```
Result delete_component(const Asn1Value &<zap>,Asn1Value &<next>)
```

This function call finds the component specified by <*zap*>, and removes it from the `Asn1Value`. The routine assigns the component that originally followed the deleted component to the `Asn1Value` specified by <*next*>. If no components remain following the one specified by <*zap*>, <*next*> is assigned an uninitialized `Asn1Value`, but the function still returns `OK`. Internally, this function performs lazy decoding, and only decodes values into components as they are needed.

Returns `NOT_OK` if the `Asn1Value` is not initialized, or if the `Asn1Value` can not be decoded into separate components.

encode_bits

```
Result encode_bits(const Asn1Tag &<tag>,
    const Octet &<val>,
    U32 <len>,
    Asn1Encoding <enc> = ENC_BER);
```

```
Result encode_bits(const Asn1Tag &<tag>,
    const DataUnit *<val>,
    U32 <len>,
    Asn1Encoding <enc> = ENC_BER);
```

This function calls copy and store the bit value specified by <*val*> into the `Asn1Value`. The length of <*val*>, in bits, is specified by <*len*>. The bit value specified by <*val*> is expected to be passed to this function as an ASN.1 bit value.

If the `Asn1Value` previously contained a value, the storage for the previous value is deallocated before the new value is stored. The tag for the `Asn1Value` is set to the value specified by the <*tag*> parameter. The encoding of the `Asn1Value` is set to the encoding specified by <*enc*>, but defaults to BER if not specified.

Returns `NOT_OK` if memory cannot be allocated to store the encoded bit value

encode_boolean

```
Result encode_boolean(const Asn1Tag &<tag>,
     const Boolean <val>,
     Asn1Encoding <enc> = ENC_BER);
```

This function call stores a copy of the boolean value specified by the <*val*> parameter in the `Asn1Value`. If the `Asn1Value` previously contained a value, the storage for the previous value is deallocated before the new value is stored. The tag for the `Asn1Value` is set to the value specified by the <*tag*> parameter. The encoding of the `Asn1Value` is set to the encoding specified by the <*enc*> parameter; if not specified, the encoding is assumed to be BER.

encode_enum

```
Result encode_enum(const Asn1Tag &<tag>,
     const I32 <val>,
     Asn1Encoding <enc> = ENC_BER);
```

This function call stores a copy of the enumerated value specified by *<val>* in the `Asn1Value`. If the `Asn1Value` previously contained a value, the storage for the previous value is deallocated before the new value is stored. The tag for the `Asn1Value` is shared with is set to the tag value specified by the tag parameter. The encoding of the `Asn1Value` is set to the encoding specified by *<enc>*, but defaults to BER if not specified.

Returns `NOT_OK` if storage cannot be allocated for the `Asn1Value`.

encode_ext

```
Result encode_ext(const Asn1Tag &<tag>,
     const Oid &<oid>
     const I32 <indirect>,*
     const DataUnit &<odes>,
     Asn1Value &<encoding>,
     Boolean <indefinite> = FALSE);
```

This function call inserts into this instance of `Asn1Value` an encoding of the `EXTERNAL` type, as defined by ISO 8824/X.208.

In that standard, the EXTERNAL type is defined as follows:

```
[UNIVERSAL 8] IMPLICIT SEQUENCE
     {direct-reference      OBJECT IDENTIFIER OPTIONAL,
      indirect-reference    INTEGER OPTIONAL,
      direct-reference      ObjectDescriptor OPTIONAL,
      encoding              CHOICE
               {single-ASN1-type   [0] ANY
                octet-aligned      [1] IMPLICIT OCTET STRING,
                arbitrary          [2] IMPLICIT BIT STRING
               }
     }
```

The arguments that are not applicable to a particular case can be empty, provided that not all of <*tag*>, <*oid*>, <*indirect*>, <*odes*>, and <*encoding*> are empty. The arguments are:

<*tag*>	Tag identifying the class of encoding
<*oid*>	Object identifier
<*indirect*>	Indirect reference
<*odes*>	Object descriptor
<*encoding*>	ASN1 encoding
<*indefinite*>	Whether this is an indefinite-length encoding

Returns `OK` if a value based on the arguments is successfully inserted into this ASN1 value.

encode_int

```
Result encode_int(const Asn1Tag &<tag>,
     const I32 <indirect>,*
     Asn1Encoding <enc> = ENC_BER);
```

This function call stores a copy of the integer value specified by the <*val*> parameter in the `Asn1Value`. If the `Asn1Value` previously contained a value, the storage for the previous value is deallocated before the new value is stored. The tag for the `Asn1Value` is shared with is set to the tag value specified by the tag parameter. The encoding of the `Asn1Value` is set to the encoding specified by the <*enc*> parameter, but defaults to `BER` if not specified.

encode_null

```
Result encode_null(const Asn1Tag &<tag>,Asn1Encoding <enc> =
ENC_BER);
```

This function call encodes a zero-length `Asn1Value` having a tag as specified by <*tag*>. The encoding of the `Asn1Value` is set to the encoding specified by <*enc*>, but defaults to `BER` if not specified. If the `Asn1Value` previously contained a value, the storage for the previous value is deallocated before the null value is stored. This function allocates storage for an AVData instance, but does not allocate storage for a DataUnit.

Returns NOT_OK if storage cannot be allocated for the AVData instance.

encode_octets

```
Result encode_octets(const Asn1Tag &<tag>,
    const Octet *<val>,
    U32 <len>,
    Asn1Encoding <enc> = ENC_BER);
```

This function call stores a copy of the octet value specified by <*val*> in the Asn1Value. The length of the octet value, in octets, is specified by <*len*>. If the Asn1Value previously contained a value, the storage for the previous value is deallocated before the new value is stored. The tag for the Asn1Value is set to the tag value specified by <*tag*>. The encoding of the Asn1Value is set to the encoding specified by <*enc*>, but defaults to BER if not specified.

Returns NOT_OK if memory cannot be allocated to store the octet value.

```
Result encode_octets(const Asn1Tag &<tag>,
    const DataUnit &<val>,
    Asn1Encoding <enc> = ENC_BER);
```

This function call shares data with the DataUnit specified by <*val*>. The tag for the Asn1Value is set to the tag value specified by <*tag*>. The encoding of the Asn1Value is set to the encoding specified by <*enc*>, but defaults to BER if not specified. If the Asn1Value previously contained a value, the storage for the previous value is deallocated before the new value is stored.

Returns NOT_OK if memory cannot be allocated to store the portion of the Asn1Value that points to the shared DataUnit.

encode_oid

```
Result encode_oid(const Asn1Tag &<tag>,
     const Octet *<val>,
     U32 <len>,
     Asn1Encoding <enc> = ENC_BER);
```

```
Result encode_oid(const Asn1Tag &<tag>,
     const Oid &<val>,
     Asn1Encoding <enc> = ENC_BER);
```

This function calls encode and store an OID as an ASN.1 `oid`. The OID can be either the address of an OID, or an octet string and length. The tag of the encoded `oid` is set to the tag specified by <*tag*>.

The encoding of the `Asn1Value` must be `BER` (`ENC_BER`), which is also the default encoding. If any other encoding is specified, the method returns `NOT_OK`. It also returns `NOT_OK` if it cannot allocate the temporary storage needed as part of the encoding process.

encode_oidstr

```
Result encode_oidstr(const Octet &<dot_str>,
     const Asn1Tag &<tag> = TAG_OID,
     Asn1Encoding <enc> = ENC_BER);
```

This function call encodes and stores an integer dot string (for example, “1.3.6.1.2.1.78”) as an ASN.1 `oid`. The tag of the encoded `oid` is set to the tag specified by <*tag*>. The encoding of the `Asn1Value` must be `BER` (`ENC_BER`), which is also the default encoding. Any other encoding causes the function to return `NOT_OK`. The function also returns `NOT_OK` if it cannot allocate the temporary storage needed as part of the encoding process.

encode_real

```
Result encode_real(const Asn1Tag &<tag>,
     const double <val>,
     Asn1Encoding <enc> = ENC_BER);
```

This function call stores a copy of the double value specified <*val*> into the Asn1Value. The tag for the Asn1Value is set to the value specified by <*tag*>. The Asn1Value's encoding is set to the encoding specified by <*enc*>, but defaults to BER if not specified. If the Asn1Value previously contained a value, the storage for the previous value is deallocated before the new value is stored.

Returns NOT_OK if storage cannot be allocated for the Asn1Value.

encode_unsigned

```
Result encode_unsigned(const Asn1Tag &<tag>,
     const U32 <val>,
     Asn1Encoding <enc> = ENC_BER);
```

This function call stores a copy of the unsigned integer value specified by the <*val*> parameter in the Asn1Value. If the Asn1Value previously contained a value, the storage for the previous value is deallocated before the new value is stored. The tag for the Asn1Value is set to the tag value specified by the <*tag*> parameter. The encoding of the Asn1Value is set to the encoding specified by the <*enc*> parameter, but defaults to BER if not specified.

```
Result encode_unsigned(const Asn1Tag &<tag>,
     Asn1Value &<comp>,
     Boolean <indefinite>;
```

This function call sets <*comp*>'s tag to <*tag*>.

encoding

```
encoding();
```

This function call returns an `Asn1Encoding` value specifying the encoding of the `Asn1Value`.

The current encoded supported types are:

`ENC_BER`	BER encoding
`ENC_UNKNOWN`	Native machine format

first_component

```
Result first_component(Asn1Value &<comp>) const
```

This function call assigns the first component in an `Asn1Value` to the `Asn1Value` specified by <*comp*>. The function returns `NOT_OK` if the `Asn1Value` is not initialized, or if the `Asn1Value` can not be decoded into separate components.

next_component

```
Result next_component(Asn1Value &<prev>,Asn1Value &<next>) const
```

This function call finds the component specified by <*prev*>, and then assigns the component that follows it to the `Asn1Value` specified by the <*next*> parameter. If no components remain following the one specified by <*prev*>, <*next*> is assigned an uninitialized `Asn1Value`, but the function still returns `OK`. Internally, this function performs lazy decoding, and only decodes values into components as the components are needed. Returns `NOT_OK` if the `Asn1Value` is not initialized, or if the `Asn1Value` can not be decoded into separate components.

retag

```
Result retag(const Asn1Tag &<tag>)
```

This function call replaces the Asn1Value's previous tag with the <*tag*> specified in the argument.

set_encoding

```
Result set_encoding(Asn1Encoding <enc>)
```

This function call sets the encoding used for the ASN.1 value. If an AVData instance has been created for the class, the encoding for the Asn1Value is set to the value specified by <*enc*> parameter. Returns OK; however, returns NOT_OK under the following circumstances:

- The Asn1Value is not initialized (that is, an AVData instance does not exist for the Asn1Value)

An encoding other than the current encoding of the Asn1Value, or ENC_UNKNOWN is specified.

size

```
U32 size()
```

This function call returns a U32 value containing the length of an encoded Asn1Value. If the Asn1Value is uninitialized, it returns a length of zero. The size returned is the length DataUnit in which the encoded value is stored

num_comps

```
U32 num_comps()
```

This function call returns the number of components of the Asn1Value.

start_const

```
Result start_construct(const Asn1Tag &<tag>)
```

```
Result start_construct(const Asn1Tag &<tag>,Asn1Encoding <enc>)
```

These function calls initiate the creation of a constructed `Asn1Value`. The tag of the constructed type is set to the value specified by <*tag*>. If the `Asn1Value` previously contained a value, the storage for the previous value is deallocated before the new value is stored. The encoding of the `Asn1Value` is set to the encoding specified by <*enc*>, but defaults to `BER` if not specified. The constructor sets a flag value to indicate whether the `Asn1Value` contains a constructed type. `start_construct` also sets the size of the value to zero.

Asn1Tag

```
const Asn1Tag &tag();
```

This function call returns a reference to the `Asn1Value`'s tag.

tagged_component

```
Result tagged_component(Asn1Value &<comp>)
```

This function call checks to see if an `Asn1Value` contains only a single component. If the `Asn1Value` contains a single, initialized component, it assigns the component to the `Asn1Value` specified by the <*comp*> parameter. If the `Asn1Value` contains more than one component, or is uninitialized, the function returns `NOT_OK`.

2.4.7 `Blockage` *class*

Inheritance: `class Blockage`

Declared in: `sched.hh`

An instance of the Blockage class is a queue of callback events that are currently blocked and must be handled when they become unblocked. The class provides methods by which you can instruct the `Blockage` function on how to deal with the callbacks on its list as they become unblocked, or force their invocation at once.

The scheduler uses an instance of Blockage to maintain the scheduler queue, called `sched_q`, which is declared as `extern`.

sched_q

Declared in: `sched.hh`

```
extern Blockage sched_q;
```

The functions `post_callback`(), `purge_callback`(), `purge_callback_handler`(), and `purge_callback_data`() use `sched_q`, the extern instance of Blockage used to manage scheduler events.

Events in the scheduler queue are blocked only until the next time the scheduler runs. The scheduler dispatches callbacks just before calling `select`() and just before returning. Callbacks are invoked in the order in which they were posted (FIFO).

Table 2-15 Blockage Public Functions

Post to a wait queue	One new callback	sleep
Invoke	First waiting callback	wakeup_first wakeup_first_now
Invoke or purge	All waiting callbacks	wakeup wakeup_now purge
Invoke or purge	Any matching callbacks	wakeup wakeup_first purge

Table 2-15 Blockage Public Functions

Invoke or purge	Any callbacks with matching handler	wakeup wakeup_first wakeup_first_now wakeup_first_call purge
Invoke or purge	Any callbacks with matching data	wakeup wakeup_first purge
Invoke or purge	Any callbacks with matching call data	wakeup_call wakeup_first_call purge purge_call
Retrieve	Number of callbacks in the queue	size

2.4.7.1 *Constructor*

```
Blockage()
```

Establishes a new instance with an empty queue of callback events.

2.4.7.2 *Destructor*

```
~Blockage()
```

Removes the Blockage instance, and purges it of its remaining callbacks, if any.

2.4.7.3 *Member Functions of* `Blockage`

post_callback

```
inline void post_callback(const Callback &<e>, Ptr <cdata>=0_)
```

Adds a `Callback` instance to the end of the scheduler's immediate callback queue, optionally with callback data <*cdata*>.

purge

```
void purge()
```

This function purges all the waiting callback events in the Blockage

```
void purge(const Callback &<e>)
```

This function purges any matching callbacks from the Blockage's wait queue.

```
void purge(CallbackHandler <handler>)
```

This function purges from the Blockage's queue those callbacks whose callback handler matches <*handler*>.

```
void purge(Ptr <data>)
```

This function purges from the Blockage's queue those callbacks whose data pointer matches <*data*>.

purge_call

```
void purge_call(Ptr <cdata>)
```

Purges from the Blockage's queue those callbacks whose callback data matches *<cdata>*.

purge_callback

```
inline void purge_callback(const Callback &<e>)
```

This function purges any matching callbacks from the scheduler queue.

purge_callback_data

```
inline void purge_callback_data(Ptr <data>)
```

This function removes all callbacks with a particular combination of handler and data value. It calls the (overloaded) operator == to do the comparison.

purge_callback_handler

```
inline void purge_callback_handler(CallbackHandler <handler>)
```

This function purges all callbacks with matching handler.

size

```
U32 size()
```

Returns the number of members in the blockage's queue.

sleep

```
void sleep(const Callback &<e>, Ptr <call_data>=0
```

Posts a callback to the Blockage’s wait queue.

wakeup

```
void wakeup()
```

This function invokes all of the Blockage’s waiting callbacks.

```
void wakeup(const Callback &<e>)
```

Invokes the callback event identified by <*e*>.

```
void wakeup(CallbackHandler <handler>)
```

Invokes any of the Blockage’s callbacks whose handler matches <*handler*>.

```
void wakeup(Ptr <data>)
```

Invokes any of the Blockage’s callbacks whose data matches <*data*>.

wakeup_call

```
void wakeup_call(Ptr <cdata>)
```

Invokes any of the Blockage’s callbacks whose callback data matches <*cdata*>.

wakeup_first

```
void wakeup_first()
```

Invokes the first of the Blockage's waiting callbacks.

```
void wakeup_first(const Callback &<e>)
```

Invokes, from the Blockage's queue, the first (earliest) callback event identified by <*e*>.

```
void wakeup_first(CallbackHandler <handler>)
```

Invokes, from the Blockage's queue, the first (earliest) callback whose handler is <*handler*>.

```
void wakeup_first(Ptr <data>)
```

Invokes, from the Blockage's queue, the first (earliest) callback whose data is <*data*>.

wakeup_first_call

```
void wakeup_first_call(Ptr <cdata>)
```

Invokes, from the Blockage's queue, the first (earliest) callback whose callback data is <*cdata*>.

wakeup_first_now

```
void wakeup_first_now()
```

Invokes immediately the first (earliest) callback event in the Blockage's queue.

wakeup_now

```
void wakeup_now()
```

Invokes immediately all waiting callback events in the Blockage’s queue.

2.4.8 `Callback` *class*

Inheritance: `class Callback`

Declared in: `callback.hh`

This class serves to post and dispatch callback events. Callbacks are declared by the application and posted to the scheduler by the various posting routines. A callback contains a function pointer specifying the handler to be called when the callback is eventually dispatched. When a callback handler is called, it is passed two pieces of data:

`user_data`	The data placed in the Callback by the application
`call_data`	Supplied by the routine dispatching the call (and can be NULL).

The scheduler keeps a queue of callbacks that should happen immediately. The scheduler routines dispatch these immediate callbacks just after the routine is entered, and again after it has dispatched all I/O and timeouts, in case the I/O or timeout callbacks scheduled any immediate callbacks themselves. Immediate callbacks occur in the order posted.

Table 2-16 Callback Public Variables

Context `context;`	The context of the callback
CallbackHandler `handler;`	The function the scheduler should call
Ptr `data;`	Data provided as argument to the callback function

Table 2-17 Callback Public Functions

Operator overloading	void*
Invoke the callback directly	exec

2.4.8.1 Constructor

```
Callback()
```

Creates a callback instance with no handler or data.

```
Callback(CallbackHandler <hand>, Ptr <d>)
```

Creates a callback instance with handler <*hand*> and data <*d*>.

2.4.8.2 Operator Overloading for `Callback`

```
friend int operator == (Callback &<el>, Callback &<e2>)
```

Returns TRUE if the two callbacks have equal data and the same context and the same callback handler; FALSE otherwise.

```
operator void*()
```

Returns a pointer to the callback's handler.

2.4.8.3 Member Functions of `Callback`

exec

```
void exec(Ptr <call_data>)
```

This function invokes the callback's handler with the callback's data and `exec`'s <*call_data*> as arguments. The callback handler is executed, in its own Try block, with `CATCHALL` set to write a record of any exceptions to `misc_stderr.print`.

CallbackHandler

Declared in: `sched.hh`

```
typedef void (*CallbackHandler)(Ptr <user_data>, Ptr <call_data>)
```

AnyHandler

Declared in: `asn1_type.hh`

```
typedef class Asn1Type(*AnyHandler)(const DataUnit &<any_name>,

const DataUnit &<defined_by_name>,

const Asn1Value &<any_value>)
```

Declares a pointer to a function that looks up the Asn1Type of <*any_value*>, for a specific combination of <*any_name*> and <*defined_by_name*>.

2.4.9 Command *Class*

Inheritance: `class Command`

`#include <pmi/command.hh>`

Data Members: No public data members declared in this class

The `Command` class is used to define unique commands that can be passed between entities. The address of the `Command` object itself is used to identify it uniquely from all other `Command` objects. A `Command` object should only be defined at the global scope, and should always be `const`.

2.4.9.1 Constructor

```
Command() ;
```

This is the constructor for the `Command` class.

2.4.9.2 *Operator*

Following is the operator for the `Command` class. It casts a `Command` instance to an unsigned integer.

```
operator U32() ;
```

2.4.10 `Config` *Class*

Inheritance: `class Config`

`#include <pmi/config.hh>`

Data Members: No public data members declared in this class.

The `Config` class is used to define the orderly management of per-program or per-MIS defaults from configuration database files.

Each instance of the `Config` class has its own mapping of default names to default values. This mapping is loaded from a file defined either in the constructor or as a result of a call to the load member function.

Each line in the file is of the form:

```
key : value
```

Initial and trailing whitespace are ignored. A comment sign ('#') can be used to indicate that the remainder of the line should be ignored. Table 2-18 lists the `Config` public functions.

Table 2-18 Config Public Functions

Function Name	**Descriptions**
`fetch`	Retrieve a default data value
`load`	Load a configuration database

2.4.10.1 Constructors

```
Config()

Config(const char *<file>)
```

These are both constructors for the `Config` class.

2.4.10.2 Destructor

```
~Config()
```

Following is the destructor for the `Config` class:

2.4.10.3 Member Functions of `Config`

This section describes the member functions of the `Config` class.

load

```
void load(const char *<env>,const char *<file>)
```

This function call loads the configuration file into the mapping table. If the environment variable <*env*> has been set, loads the file corresponding to <*env*>'s value. If it is not set, the file `$EM_MIS_HOME/`<*file*> is used. If `$EM_MIS_HOME` is not set, the local directory copy of <*file*> is used.

```
void load(const char *<file>)
```

This function call loads the configuration file <*file*> into the mapping table.

fetch

```
const char *fetch(const char *<key>,const char *<dflt> = 0)
```

This function call looks up a mapping in the table. If the mapping does not exist, the value <*dflt*> is returned. Otherwise, returns a pointer to the string representing the value.

2.4.11 `DataUnit` *Class*

Inheritance: `class DataUnit`

`#include <pmi/du.hh>`

Data Members: No public data members declared in this class

The `DataUnit` class is a basic storage unit for data. Each instance of a `DataUnit` contains attribute information (defined by the `DataUnit` class definition) which includes a pointer to a chunk of memory allocated by the `DataUnit` for the storage of data. The allocated storage provided by the `DataUnit` is typically allocated when the `DataUnit` is instantiated, but can also be reallocated when a value is assigned to a `DataUnit`. A `DataUnit` can be referenced (or shared) by a number of attributes or variables. Each `DataUnit` maintains a count of the number of attributes or variables that reference it. When a `DataUnit` is created, its reference count is set to one. The count is incremented each time another value references it, and is decremented each time another value dereferences it. When the reference count goes to zero, the storage allocated for the `DataUnit` is deallocated. Each `DataUnit` also keeps track of the amount of dynamic memory it has allocated for storing data. Table 2-19 lists the `DataUnit` class public functions.

Table 2-19 DataUnit Public Functions

Function Name	**Descriptions**
Octet* * [] = == !=	Operator overloading
size	Description of the DataUnit

Table 2-19 DataUnit Public Functions

Function Name	**Descriptions**
cmp	Compare two DataUnits for sorting
copy catenate	Create a new data unit from an existing one, or from two existing ones
fragment copyin copyout	Set or get a portion of the DataUnit
chp	Data as a character string

2.4.11.1 *Constructors*

This sections describes the constructors for the `DataUnit` class.

```
DataUnit();
```

This constructor creates an instance of a `DataUnit`, but does not allocate any memory for storing data. The size of the `DataUnit` is set to zero.

```
DataUnit(U32 <size>, Octet *<data> = 0);
```

This constructor creates an instance of a `DataUnit` and allocates the number of bytes of memory specified by <*size*> for storing data. If the data specified by the Octet *<*data*> parameter is non-null, <*size*> bytes of data are copied from the octet string specified by <*data*> to the newly allocated storage, and the size of the `DataUnit` is set to <*size*>.

```
DataUnit(const char *<str>)
```

This constructor creates an instance of `DataUnit` from a NULL terminated string <*str*>. The DataUnit instance points to the same memory as <*str*> and is assumed that the memory pointed by <*str*> is valid and is *not* changed during the lifetime of the DataUnit instance being created. In other words, unlike DataUnit `(char *`<*str*>`)`, this constructor does not allocate new memory and copy contents of <str> into it. (See `DataUnit(char *<str>)`.

```
DataUnit(const DataUnit &<du>);
```

This constructor creates an instance of `DataUnit` but does not allocate any storage for it. The new `DataUnit` simply points at the storage allocated for the `DataUnit` identified by <*du*>. The reference count for the shared memory is incremented to reflect the additional `DataUnit` that is now pointing to it.

```
DataUnit(U32 <size>, const Octet *<data>);
```

This constructor creates an instance of `DataUnit` which points at the storage associated with <*data*>. Its length is indicated by <*size*>.

```
DataUnit(U32 <size>, int <fill>);
```

This constructor creates an instance of `DataUnit` and allocate storage of length <*size*> bytes. Each byte of the allocated storage is assigned the value <*fill*>.

```
DataUnit(char *<str>);
```

This constructor creates an instance of `DataUnit` with storage sufficient to hold a duplicate of <*str*> (assumed to be null-terminated), which is copied to the `DataUnit`. It makes a *copy* of <*str*>, as opposed to `DataUnit(const char *<str>)`, in which the `DataUnit` points to the same area of memory as const char *<*str*>; this assumes that the string could change.

2.4.11.2 Destructor

```
~DataUnit();
```

This destructor decrements the reference count for the (potentially) shared memory region used to store data for the `DataUnit`. If the reference count reaches zero, the memory used to store data is deallocated.

2.4.11.3 Operator Overloading for DataUnit

This sections describes the operators for the `DataUnit` class.

```
Octet *() const;
```

This operator returns a pointer to the first Octet of allocated storage in the `DataUnit`.

```
DataUnit &operator = (const DataUnit &<du>);
```

This operator returns a reference to the storage allocated for the `DataUnit` specified by <*du*>. The reference count for the storage is incremented. If the `DataUnit (*this)` previously pointed to allocated storage, the reference count for that storage is decremented, and if the reference count reaches zero, the storage is deallocated.

```
int operator == (const char *<str>) const;
```

This operator compares the contents of the `DataUnit` with <*str*>. If they are the same, the function returns 1. If they differ, it returns 0.

```
Octet &operator *() const;
```

This operator returns a reference to the first Octet of allocated storage in the `DataUnit`. This operator triggers an assertion failure if the `DataUnit` is uninitialized.

```
Octet &operator [] (U32 <index>) const;
```

This operator returns a reference to the Octet of allocated storage that is <*index*> bytes past the first byte of storage allocated for the `DataUnit`. The first byte of allocated storage would have an index value of 0; the second byte, an index value of 1; the third byte, an index value of 2; and so on. This operator triggers an assertion failure if the value specified by the index parameter is greater then the size of the allocated storage, or if the `DataUnit` is uninitialized.

```
friend int operator == (const DataUnit &<du1>, const DataUnit &<du2>)
```

This operator compares the two `DataUnit`'s specified by <*du1*> and <*du2*> and returns a nonzero integer if the two are equivalent.

```
friend int operator != (const DataUnit &<du1>, const DataUnit &<du2>)
```

This operator compares the two `DataUnit`'s specified by <*du1*> and <*du2*> and returns a nonzero integer if the two are not equivalent.

```
int operator != (const char *<str>) const;
```

This operator compares the contents of the `DataUnit` with <*str*>. If they are the same, the function returns 0. If they differ, it returns 1.

2.4.11.4 Member Functions of `DataUnit`

This sections describes the member functions of the `DataUnit` class.

catenate

```
DataUnit catenate(const DataUnit &<du1>, const DataUnit &<du2>)
```

This function call constructs a `DataUnit` and allocates it dynamic storage equal to the size of the dynamic storage of *<du1>*, plus the size of the dynamic storage for *<du2>*. The `catenate` function then copies the allocated storage from *<du1>*, followed by the storage from *<du2>*, to the newly allocated storage. The function then returns the new `DataUnit`.

chp

```
char* chp() const;
```

This function call returns a pointer to a character string containing a representation of the `DataUnit`'s data. The representation does not have a trailing newline character.

cmp

```
friend int cmp(const DataUnit &<du1>, const DataUnit &<du2>)
```

This function call performs a machine-dependent comparison of the two `DataUnit`'s specified by *<du1>* and *<du2>*. The function returns a result in the same way as `strcmp`: that is, the result is 0 if the two `DataUnit`'s compare equal, -1 if*<du1>* is lexicographically less than *<du2>*, and 1 if *<du1>* is lexicographically greater than *<du2>*.

copy

```
void copy (U32 <pos>
    U32 <ln>,
    const DataUnit &<du>,
    U32 <du_pos>)
```

This function call copies data from the `DataUnit` specified by <*du*> to the `DataUnit` specified by `*this`. Data is copied from <*du*> starting from an offset into <*du*>'s allocated storage specified by <*du_pos*>. The data is copied to `*this` starting at an offset into `*this`'s allocated storage specified by <*pos*>. The number of octets copied is specified by <*ln*>. This function triggers an assertion failure if the number of octets to copy, specified by <*ln*>, plus the starting point specified by either <*du_pos*> or <*pos*>, is greater than the storage allocated for either <*du*> or `*this`, respectively.

copyin

```
void copyin (const Octet *<data>,U32 <pos>,U32 <ln>);
```

This function call copies data from the location specified by the Octet *<*data*> parameter, into the dynamic memory allocated for the `DataUnit`. Data is copied into the allocated storage starting at a point that is <*pos*> bytes past the start of the allocated storage. The number of octets to copy is specified by <*ln*>. The function triggers an assertion failure if the starting position (specified by <*pos*>), plus the length to copy (specified by <*ln*>) is greater than the number of octets of storage allocated for the `DataUnit`.

copyout

```
void copyout (Octet *<data>
    U32 <pos>,
    U32 <ins>) const;
```

This function call copies data from the `DataUnit` to location specified by the Octet *<*data*> parameter. Data is copied from the allocated storage of the `DataUnit` starting at a point that is <*pos*> bytes past the start of the allocated storage. The number of octets to copy is specified by <*ln*>. The function triggers an assertion failure if the starting position (specified by <*pos*>), plus the length to copy (specified by <*ln*>) is greater than the number of octets of storage allocated for the `DataUnit`.

equiv

```
static int equiv (Ptr <n1>,Ptr <n2>);
```

This function call is used to compare two `DataUnit` pointers in a manner consistent with the declaration of the Hash macros.

fragment

```
DataUnit fragment(U32 <st>,U32 <ln>) const;
```

This function call returns a `DataUnit` that points to a portion (or fragment) of the dynamic storage allocated for the `DataUnit` specified by `*this`. The fragment shares data with a portion of the storage allocated for the original `DataUnit`. The fragment starts <*st*> octets after the start of the allocated storage and extends for <*ln*> octets.

This function triggers an assertion failure either of the following occur:

- the start of the fragment, specified by <*st*>, plus the length of the fragment, specified by <*ln*>, is greater than the length of the allocated storage of `DataUnit` specified by `*this`
- the `DataUnit`, specified by `*this`, is not initialized but <*st*> or <*ln*> is nonzero.

hash

```
static U32 hash(DataUnit *<du>)
```

This function call can be used when declaring Hash macros where a `DataUnit` is used as the key.

size

```
U32 size() const
```

This function call returns the number of Octets of the storage allocated for the `DataUnit`.

unshare

```
void unshare()
```

This function call makes memory storage associated with a `DataUnit` not shared. Creates a private copy if it is currently shared.

2.4.12 `Dictionary` *Class*

Declared in: `dict.hh`

The `Dictionary` class is the set of facilities that are provided to C++ classes created using the `Dictionarydeclare` macro.Dictionary tables are accessed in two ways: as indexable elements and using a key/value paradigm. Implicit in the definition of the dictionary is the `lookup`() function that provides the mechanism to map an arbitrary key value to an index in the array.

```
#define Dictionary (K, T)
    name3(K,T,Dict)
#define Dictionarydeclare(K,T)
#define Hdict(K,T)
    name3(K,T,Hdcit)
#define Hdictdeclare(K,T)
#define Hrefdict(K,T)
    name3(K,T,Hrefdict)
#define Hrefdictdeclare(K,T)
```

Table 2-20 Dictionary Protected Variables

const T ***table_**;	The internal table of elements
U32 **len_**;	The number of elements in the table

Table 2-21 Dictionary Functions

lookup	Look up an element
num_elems	Return the number of elements
position	Return the index of an element
table	Return a pointer to the table

2.4.12.1 Constructor

```
Dictionary(K,T)(const T *<t>,

U32 <ne>)
```

This constructor initializes a table pointer to <*t*> and assigns a length of <*ne*>.

2.4.12.2 Destructor

```
~Dictionary(K,T)()
```

The destructor leaves the memory associated with the table unchanged while removing the pointer referencing it.

2.4.12.3 *Operator Overloading for* `Dictionary(K,T)`

```
const T&operator[](U32 <pos>))
```

This function returns the nth element of the table where <*pos*> equals <*n*>.

```
const T*lookup(const K &<key>)
```

This function, which must be implemented for each <*K*> and <*T*> that are defined for a usage of `Dictionarydeclare`(), returns a pointer to a table element of type <*T*> based on a key of type <*K*>.

```
U32 num_elems() const;
```

This function returns the number of elements in the table.

```
U32 position(const T *<t>) const;
```

Given a pointer to the table element <*t*>, this function returns the index to the table.

```
const T *table()
```

This function returns a pointer to the table of elements managed within the Dictionary.

2.4.13 `Hashdeclare` *Macro*

Declared in: `hash.hh`

```
Hashdeclare(<keytype>, <valtype>, <hashfun>, <eqfun>, <delkey>, <delval>)
```

`Hashdeclare` is a macro used to create a hash table class that is specific to the types of its keys and values. To refer to the class thus created, use the `Hash(`*<keytype>*`,` *<valtype>*`)` macro (described below).

The arguments *<keytype>* and *<valtype>* are the names of declared types or classes. The arguments *<hashfun>*, *<eqfun>*, *<delkey>*, and *<delval>* are the names of functions required by the constructor for the `HashImpl` class.

Because the macro invokes functions defined in the `HashImpl` class, the class it produces is in a sense "derived" from `HashImpl`. Each of the member functions defined by `Hashdeclare` has a corresponding function of the same name defined by the `HashImpl` class. The macro and the `HashImpl` class also provide alternate calls to calculate the hashed values for you when you don't want to do it yourself.

2.4.13.1 Member Functions for `Hashdeclare` *Macro*

strhash

Declared in: `hash.hh`

```
U32 strhash(const char *<str>)
```

The `strhash`() function is a sample hashing function for null-terminated strings. It could be used to calculate hash values for passing to the member functions of `HashImpl`, or its name could be passed to the `Hashdeclare` macro as the hash function for keys of the appropriate type.

2.4.14 `HashImpl` *Class*

Inheritance: `class HashImpl`

Declared in: `hash.hh`

The `HashImpl` class provides a template class used by the `Hashdeclare`() macro to implement a dynamically growing hash table. It maintains the private data structures that constitute the hash table.

The HashImpl class is intended to be somewhat primitive; it is normally hidden behind the HashDeclare macro. The HashImpl member functions have no idea what they're actually managing. All keys and values are passed to them as void* values. For this reason, when the constructor for HashImpl is called, you must supply pointers to three functions:

- A function that takes two arguments of type void* and returns whether the two values are to be considered equal.
- A function (which might be NULL) to invoke when keys are to be destroyed. It is passed a single void* value.
- A function (which might also be NULL) to invoke when values are to be destroyed. It is also passed a single void* value.

These last two functions, if specified, are called whenever key/value pairs need to be deleted from the hash table, including when the entire table is destroyed.

It is often the case that the same key is used several times in a row. For this reason, all of the member functions that take a key argument also take an argument that is the hashed value of the key. The hash value need be recomputed only when the key changes. You can call any hash function you like, but it should return an unsigned 32-bit integer, hopefully with a uniformly pseudorandom distribution in the lowest <*n*> bits (where <*n*> is the number of bits necessary to index into a hash table big enough to hold the maximum number of entries, more or less). A sample hash function for null-terminated strings is supplied; see the member function, strhash().

In order for a key to match an entry in the table, its hashed value must first match the hashed value stored in the table entry. You could conceivably make use of this in a smart hash function to make otherwise identical keys behave as if they're different keys.

Variables: No public variables declared in this class.

Table 2-22 HashImpl Public Functions

Public	
Fetch a value from the hash table	fetch next

Store a key and value in the hash table	store
Destroy a key-value pair	destroy iterate

2.4.14.1 Constructor

```
HashImpl(int (*<eqfun>)(),
    void (*<delkey>)(),
    void (*<delval>)())
     { <heads> = 0;
     <maxbucket> = 0;
     <bucketsused> = 0;
     <ef> = <eqfun>;
     <df> = <delfun>;
```

2.4.14.2 Destructor

```
~HashImpl()
    {void *<key>;
     void *<val>;
    U32 <hashval>;
      if (<dkf> || <dvf>)
    {iterate();
      while (next(<key>, <val>, <hashval>))
    {destroy(<key>, <hashval>);
    }
     }
    }
```

2.4.14.3 Member Functions of `HashImpl`

Apart from `iterate`(), the member functions of `HashImpl` are provided as implementations of the member functions of the specific hash class created by a use of the `Hashdeclare` macro.

destroy

```
void *destroy(void *<key>, U32 <hashval>)
```

Deletes the key / value pair from the hash table. Returns the deleted value (which might have been destroyed, if a <*delval*> function was specified to the constructor).

fetch

```
void *fetch(void *<key>, U32 <hashval>)
```

Returns a pointer to the value associated with <*key*> and <*hashval*>.

iterate

```
void iterate()
```

Used internally by the class constructed by the `Hashdeclare` macro to reset the hash table's built-in iterator to the beginning.

next

```
Boolean next(void* &<key>,
    void* &<val>,
    U32 &<hashval>)
```

Returns TRUE if more entries remain in the hash table. Sets `void*` variables for the next key / value pair from the hash table, according to the table's built-in iterator. It also sets the corresponding <*hashval*> variable. It is legal to destroy the current value.

store

```
void *store(void *<key>,
    U32 <hashval>,
    void *<value>);"
```

Stores a pointer to the value associated with the specified <*key*> and <*hashval*>, and returns that value. Any previous value associated with that <*key*> and <*hashval*> are destroyed. Returns the new value.

strhash

Declared in: `hash.hh`

```
U32 strhash(const char *<str>)
```

The `strhash`() function is a sample hashing function for null-terminated strings. It could be used to calculate hash values for passing to the member functions of `HashImpl`, or its name could be passed to the `Hashdeclare` macro as the hash function for keys of the appropriate type.

2.4.15 `Hdict` *Class*

Declared in: `dict.hh`

The `Hdict` class is the set of facilities provided to C++ classes created using the `Hdictdeclare`() macro. It utilizes the Hash mechanism to manage the lookup of elements in its table. It assumes that an appropriate `Arraydeclare(T)` and `Hashdeclare(K,T)` have preceded its declaration.

Table 2-23 Hdict Protected Variables

Hash(K,T) ***_hash**;	The internal hash table of elements
Array(T) **_array**;	The array of elements

Table 2-24 Hdict Public Functions

lookup	Lookup an element
num_elems	Return the number of elements
table	Return a pointer to the table
position	Return the index of an element
set	Initialize the array and the hash table
sfun	Return a key pointer

2.4.15.1 *Constructors*

```
Hdict(K,T)
```

This constructor creates an empty table.

```
Hdict(K,T)(Array(T)&<a>)
```

This constructor creates a hashing dictionary by stealing the array associated with <*a*> and initializing an internal hash table.

2.4.15.2 *Destructor*

```
~Hdict(K,T)()
```

The destructor deletes the memory associated with the hash table and array.

2.4.15.3 *Operator Overloading for* `Hdict(K,T)`

```
const T&operator[](U32 <pos>))
```

This function returns the nth element of the table where <*n*> equals <*pos*>.

2.4.16 `Oid` Class

Inheritance: `class Oid : public DataUnit`

`#include <pmi/oid.hh>`

Data Members: No public data members declared in this class

The `Oid` class defines a container for an object identifier. Each object identifier is a sequence of numbers that identify an object by an integer denoting its assigned position at each level of a tree-structured registry of object names.

Many of the properties of the `Oid` implementation (for example, data storage, sharing) are derived from `DataUnit`. Table 2-25 lists the `Oid` class public functions.

Table 2-25 Oid Public Functions

Function Name	**Descriptions**
=	Operator overloading
copy_oid format print	Extract the entire oid
num_ids	Length of the oid
get_id	Extract a given id from within the oid
add_id add_last_id append	Add to the oid
is_same_prefix	Compare two ids

2.4.16.1 Constructors

This sections describes the constructors of the `Oid` class.

```
Oid()
```

This constructor constructs an empty `Oid`.

```
Oid(U32 <size>,const Octet *<data> = 0)
Oid(const DataUnit &<du>)
```

```
Oid(const Oid &<oid>)
```

```
Oid(const char *<str>)
```

Each of the preceding constructors construct an `Oid` from a particular representation of the list of IDs that comprise it.

```
Oid(U32 <size>,

    U32 <first_id>,

    U32 <second_id>,

    U32 &<place>)
```

This constructor constructs an object identifier whose first identifier is <*first_id*> and whose second identifier is <*second_id*>. The size of the allocated memory is <*size*>. The returned value <*place*> is used when appending new identifiers to the object identifier string (see `add_id`).

2.4.16.2 *Operator Overloading of* `Oid`

This section describes the operators for the `Oid` class.

```
const Oid &operator = (const Oid &<oid>)
```

This operator shares the data storage associated with <*oid*>.

2.4.16.3 *Member Functions of* `Oid`

This section describes the member functions of the `Oid` class.

add_in

```
Result add_in(U32 <id>,U32 &<place>)
```

This function call appends <*id*> to the `Oid`, and sets <*place*> to the current size (and hence the position at which <*id*> was inserted).

The `Oid(U32, U32, U32, U32)` constructor initializes <*place*>.

If need be, `add_id` extends the `Oid` to accommodate the new <*id*>. (You could use the `Oid`(<*size*>) constructor to pre-extend the `Oid` instance to the desired size.)

add_last_id

```
Result add_last_id(U32 <id>,U32 &<place>)
```

This function call appends <*id*> to the become the last id in the `Oid` instance. Any additional storage is released.

append

```
static Result append(Oid &<front>,
    Oid &<back>,
    U32 <front_place>)
```

This function call modifies <*front*> by appending the IDs contained in <*back*>. Returns `OK` if this is completed. Sets <*front_place*> to the last-written position in <*front*> (which is also the numbers of ids in <*front*> when the operation is complete).

copy_oid

```
static Result copy_oid(const Oid &<src>,Oid &<dest>)
```

This function call makes <*dest*> the same as <*src*>.

format

```
Result format(char *<buf>,U32 <buf_len>) const
```

This function call writes to <*buf*> a string of length <*buf_len*> containing the formatted representation of the `Oid`. In the string, each ID is represent as decimal digits, and successive IDs are separated by a dot.

get_id

```
Result get_id(U32 <id_num>,
     U32 &<id>,
     U32 &<place>) const
```

This function call extracts from the `Oid` its <*id_num*>'th id and stores it in `id`. (The first `id` is considered to be at position 1). To speed subsequent access, stores at <*place*> the current position within the `Oid`, and initialize it to zero before the first use.

is_same_prefix

```
Result is_same_prefix(const Oid &<o1>,const Oid &<o2>)
```

This function call compares two `Oid`s to determine whether the one with the shorter sequence of IDs is a prefix of the other, and returns true if it is. (If both `Oid`s contain the same number of IDs, this boils down to asking whether they are equal.)

If the function returns false, it means some internal error has occurred. If no error is returned, <*o1*> and <*o2*> do not have the same prefix.

num_ids

```
U32 num_ids() const
```

This function call returns the number of IDs in the `Oid` instance.

print

```
void print(FILE *<fp>) const;
```

```
void print(DEBUG &<deb> = misc_stdout) const;
```

```
void print(char *<c>) const;
```

Each of the preceding function calls print a formatted record of the `Oid`, appending it either to the file whose pointer is <*fp*>, or the debug stream <*deb*>, or to the character string pointed to by <*c*>.

2.4.16.4 `Asn1TypeDefinedType` *Declarations*

Following is a list of the `Asn1TypeDefinedType` declarations:

```
extern Asn1TypeDefinedType NumericStringType;
extern Asn1TypeDefinedType PrintableStringType;
extern Asn1TypeDefinedType TeletexStringType;
extern Asn1TypeDefinedType VideotexStringType;
extern Asn1TypeDefinedType VisibleStringType;
extern Asn1TypeDefinedType IA5StringType;
extern Asn1TypeDefinedType GraphicStringType;
extern Asn1TypeDefinedType GeneralStringType;

extern Asn1TypeDefinedType GeneralizedTimeType;
extern Asn1TypeDefinedType UTCTimeType;
extern Asn1TypeDefinedType EXTERNALType;
extern Asn1TypeDefinedType ObjectDescriptorType;
```

2.4.16.5 Asn1SubTypeKind

Following is the `Asn1SubTypeKind` declaration:

```
enum Asn1SubTypeKind
  {ASK_NONE,
   ASK_SINGLE,
   ASK_CONTAINED,
   ASK_RANGE,
   ASK_PERMITTED,
   ASK_SIZE,
   ASK_INNER_SINGLE,
   ASK_INNER_MULTIPLE
  } ;
```

2.4.16.6 Asn1SubTypeSize

Following is the `Asn1SubTypeSize` declaration:

```
typedef Asn1SubTypeSize Asn1SubTypePermitted;
typedef Asn1SubTypeSize Asn1SubTypeInnerSingle;
```

2.4.16.7 Asn1Kind

Following is the Asn1Kind declaration:

```
enum Asn1Kind {
  AK_NONE,
  AK_BOOLEAN,
  AK_INTEGER,
  AK_BIT_STRING,
  AK_OCTET_STRING,
  AK_NULL,
  AK_SEQUENCE,
  AK_SEQUENCE_OF,
  AK_SET,
  AK_SET_OF,
  AK_CHOICE,
  AK_SELECTION,
  AK_TAGGED,
  AK_ANY,
  AK_OBJECT_IDENTIFIER,
  AK_ENUMERATED,
  AK_REAL,
  AK_SUBTYPE,
  AK_DEFINED_TYPE
};
```

2.4.16.8 Asn1TypeE

Following is the Asn1TypeE declaration:

```
typedef Asn1TypeE Asn1TypeSeqOf;
typedef Asn1TypeE Asn1TypeSetOf;
```

2.4.16.9 Asn1TypeEL

Following is the Asn1TypeEL declaration:

```
typedef Asn1TypeEL Asn1TypeSeq;
typedef Asn1TypeEL Asn1TypeSet;
```

2.4.16.10 Asn1TypeNN

Following is the Asn1TypeNN declaration:

```
typedef Asn1TypeNN Asn1TypeInt;
typedef Asn1TypeNN Asn1TypeEnum;
typedef Asn1TypeNN Asn1TypeBitStr;
```

2.4.16.11 Asn1TagClass

Following is the Asn1TagClass declaration:

```
declared in: /opt/SUNWconn/em/include/pmi/asn1_val.hh
typedef enum Asn1TagClass
      {CLASS_UNIV,
       CLASS_APPL,
       CLASS_CONT,
       CLASS_PRIV } ;
```

2.4.16.12 Asn1Tagging

Following is the Asn1Tagging declaration:

```
// declared in: /opt/SUNWconn/em/include/pmi/asn1_val.hh
typedef enum Asn1Tagging
      {TAG_EXPLICIT,
       TAG_IMPLICIT,
      } ;
```

2.4.17 Timer *Class*

Inheritance: class Timer

#include <pmi/timer.hh>

An event that is scheduled to occur after a specific interval is represented as an instance of the Timer class. Timer events are posted or removed from the timer queue by the functions `post_timer()`, `purge_timer()`, `purge_timer_data()`, and `purge_timer_handler()`. Each function is declared in sched.hh.

A `Timer` event is not dispatched before the specified interval has elapsed. However, it might have to wait longer if other processing has to happen when its interval has elapsed.

When the callback does occur, the timer automatically reposts itself if you specified a *reload time* in the Timer. (By default, the reload time is 0, meaning no reposting.) Both the invocation time and the reload time are specified in milliseconds.

The invocation time is the interval from the time at which you call the `post_timer`() routine. The reload time is the interval from the originally scheduled time (not from the time at which the callback was actually dispatched). If the callback is delayed to the point that the reload interval has already elapsed, the scheduler simply skips a reload for an interval that has now elapsed, and schedules the reload for the next upcoming multiple of the reload interval.

Timers only guarantee that the callback isn't run too early. There's no guarantee about running too late, so don't try to use this to control real-time interactions.

Timers can be purged from the timer queue by matching on the enqueued object's handler, data, or both.

2.4.17.1 Default Constructor

```
Timer()
```

Resets the time and reloads the timer.

2.4.17.2 Constructor

```
Timer(MTime t, MTime re, CallbackHandler hand, Ptr d)
```

Resets the time and reloads the timer. This constructor has four arguments. The time (*t*) between now and the first callback; the reload time (*re*) between callbacks; the callback pointer (*hand*) to the functions which is executed; and the pointer (*d*) to the user data which is passed to the callback hand.

2.4.17.3 Operator

```
friend int operator==(const Timer &t1, const Timer &t2)
```

Compares the timers to determine whether they have the same callback function or not.

2.4.17.4 Member Functions for Timer Class

post_timer

```
void post_timer(Timer &)
```

Posts a timer event into the timer queue.

purge_timer

```
void purge_timer(Timer &)
```

Purges from the timer queue any matching timer events.

purge_timer_data

```
void purge_timer_data(Ptr <data>);
```

Purges from the timer queue any timer events whose data matches <*data*>.

purge_timer_handler

```
void purge_timer_handler(CallbackHandler <handler>);
```

Purges from the timer queue any timer events whose handler matches <*handler*>.

High-Level PMI 3

3.1 Introduction

The Solstice EM product provides a Portable Management Interface (PMI) with a suite of classes and member functions that provide effective access to the Solstice EM Management Information Server (MIS) without requiring detailed specification of the underlying MIS or mechanism. For most applications, the high-level usage of the PMI is sufficient for all interaction with the Solstice EM MIS.

This chapter includes the following classes:

Table 3-1 High-Level PMI Classes

Class	**Description**
Album Class	Represents a set of related objects
AlbumImage Class	Represents the state of an iterater
AppTarget Class	Represents target applications
AuthApps Class	Used when you need to know which applications a user is authorized to use
AuthFeatures Class	Used when you need to implement the feature level access control in your application
Coder Class	Represents a pair of methods for encoding and decoding values
CurrentEvent Class	Represents an event
Error Class	Stores details of errors related to an object instance
Image Class	Represents an actual or potential object in an MIS' framework
Morf Class	Represents a unit of data
PasswordTty Class	Implements the TTY based password query mechanism
Platform Class	Represents a potential or actual connect to a MIS
Syntax Class	Represents a type
Waiter Class	Represents an ongoing asynchronous operation

3.2 Design Objectives

The PMI seeks to balance two contrasting goals:

- *Locational transparency.* It should be convenient to write an application without having to know where and how its objects are stored.
- *Locational flexibility.* It should be convenient to write an application that takes specific account of where and how its objects are stored.

To achieve locational transparency, we need:

- MIS independence

- Automatic propagation of *event sieves* from the application to the MIS.
- Automatic *caching* of various sorts of data in the application process.

To achieve locational flexibility, we need:

- Access to (some of) the low level primitives upon which the high level usage of the PMI is built, in such fashion that using the low level primitives of the PMI does not confuse the high level primitives of the PMI.

3.3 Object Management Model

3.3.1 Manipulating Objects

Low-level routines for manipulating objects tend to manage objects by sending them messages and waiting for replies. In CMIS, for instance, there are messages to create and delete objects, to get and set attributes, and to perform various actions or to signal events. The PMI replaces these notions with a model in which objects appear (as far as possible) to be local. The remote object is tracked in the application process by a local class object called an image, which acts as a surrogate for the remote object and tracks where it is and what it's doing.

3.3.2 Naming Objects

Objects are named by starting from a known starting point in a tree and traversing a map of containment relationships. The object's name is formed by concatenating the "key" for each of the steps in the traversal, with slashes between the components. To find an object's container, you have only to strip off the final component of the object's name.

An object name that begins without a slash is a local distinguished name and is under the local root. In the case of the Solstice EM MIS, the local root is `/systemID`. An object name that begins with a slash is a distinguished name, and refers to an object that is global (in the literal sense). Names used in the PMI can comprise a superset of the OSI naming tree. Names that are not part of the OSI naming tree must not conflict with the OSI naming tree.

Within the application, any object can also have a nickname. Nicknames offer a convenient way for you to program readably, and also provide a level of indirection that helps you in your quest for abstraction.

3.3.3 Relationships Between Objects

The relationships between objects are represented using album objects.

We often think of undirected sibling-like relationships as a form of set membership. The album contains a set of `image`s, so membership of an image in an album can model the inclusion of objects in a mathematical set. The application can build up an album by enumerating the `image`s that it wants to include, much as a mathematician might build a set by enumeration.

Other relationships are of a directed nature. The PMI models these relationships as albums built by a derivational rule rather than by enumeration. A simple derivation might form the set of objects that are “children” of another object. A more complex derivation might examine all the objects in another album, select a subset of them, and for each of that subset specify a relational attribute that defines a new set of objects.

3.3.4 Managing Notifications

An image (or an album of `image`s) can alert the application when a change of state in which the application has expressed an interest occurs. The PMI transmits the event to the application by invoking the callback function that the application registered earlier.

3.3.5 Managing Data Types

Each image knows the attributes that a given object class supports. It also knows the type of each attribute. That lets the application deal with the object on a purely textual basis if it chooses.

The language in which textual data is expressed looks much like ASN.1 textual data. Scoping is indicated by curly braces, choice names are delimited by a colon, and so on. This approach was decided upon because ASN.1 specification is complete and mature, at least when compared with other abstract syntax notations. You can represent other abstract notations with this data language, but it just maps most easily onto ASN.1.

Sometimes the application needs to deal with data apart from the definition of any particular object. While this can be done using text, as mentioned above, it is sometimes more convenient to pass around encoded data. The PMI supports the notion of typed, encoded data. Such an object is called a `Morf`. You can think of a Morf as a bare attribute without any associated object. It knows its type and the associated syntax, so you can convert its value to and from textual representation, just as you would an attribute of an image object.

3.3.6 Object Schema Management

The application program can discover various facts about the object class and its various attributes by using the get_prop() and get_attr_prop() functions. These routines work much like the UNIX getenv call, but acquire their information from the MIS or from some associated repository of metadata. We'll call these attribute-like facts "properties" to avoid confusing them with the ordinary attributes contained in a managed object instance.

3.3.6.1 Where Properties Occur

Properties occur among attributes used by the Image class, and in the arguments or results of methods of the Image, Album, and Platform classes. Table 3-2 summarizes where various properties occur. See also the descriptions of:

- get_prop, under the description of the Album class, which starts on page 3-17
- get_prop, under the description of the Image class, which starts on page 3-59
- get_attr_prop, also under the description of the Image class

Table 3-2 Properties in Album, Image, and Platform

Property	**Image Attribute**	**Image**	**Album**	**Platform**
ACCESS			X	
APPLICATION_INSTANCE_NAME				X
APPLICATION_TYPE				X
AUTOIMAGE			X	
DEFAULT_ALLOWED	X			
DEFAULT_TIMEOUT				X
DERIVATION			X	

Table 3-2 Properties in Album, Image, and Platform

Property	Image Attribute	Image	Album	Platform
EXCLUDE_ALLOWED	X			
EXISTS	X	X		
IGNORE_ALLOWED	X			
LOCATION				X
MOD_PENDING	X			
MODIFIABLE	X			
NICKNAME		X	X	
NICKNAME_IS_PERMANENT		X		
OBJCLASS		X		
OBJNAME		X		
OWNERSHIP			X	
PLATFORM_NICKNAME				X
PLATFORM_OBJNAME				X
PLATFORM_TYPE				X
REPLACE_ALLOWED	X			
STATE		X	X	X
TRACKMODE	X	X	X	

3.3.7 Filtering as an Aspect of Album Derivation

The `Album` function `set_derivation()` (or its more general form `set_prop`) can specify a derivation that includes a CMIS filter. The derivation specifies three items, in this order:

1. Object name
2. Scope
3. Filter

A slash separates the scope from the object name. If there is a filter, a slash separates the scope from the filter.

See the appendix on scoping and filtering in the *Solstice Enterprise Manager Reference Manual* for detailed information on the uses of scoping and filtering in Solstice EM.

3.3.7.1 *Object Name*

An object name is a distinguished name in slash form. The object name specifies the base object for scoping. This base object is not necessarily one of the objects in the album.

It is permissible to omit the object name. If you omit the object name, the system object is assumed. Note that if you omit the object name, you should not insert the slash that would separate the name from the derivation. If you write `/ALL`, you are indicating an object name of `/` (indicating the system object), followed by the scope `ALL`.

3.3.7.2 *Scope*

The scope can be any of the following:

Table 3-3 Scoping Parameters

Parameter	**Description**
`ALL`	All descendants of named object, including object.
`LV` <*n*>	Level <*n*> descendants only. Children are at LV(1). Grandchildren are at LV(2). The base object is at LV(0).
`TO` <*n*>	All levels down to level <*n*>, including object
`*`	Short for LV(1), i.e. children only
`*/*`	`Short` for LV(2), i.e. grandchildren only.
`*/*/*`	`Short` for LV(3), i.e. great-grandchildren only.

If omitted, the scope defaults to the base object only. Note that the asterisk forms are short for LV(*n*), not TO(*n*).

3.3.7.3 Filter

The filter is currently specified in raw ASN.1 format. The definition of the filter goes inside the parentheses that follow `CMISFilter`. This syntax makes `CMISFilter` look like a function. In fact, it is not a function; the use of parentheses is simply a convention for delimiting the filter definition.

Omitting the filter has the same effect as a filter whose definition is TRUE, meaning "include everything specified by the scope." The definition of a legal `CMISFilter` is specified in `etc/asn1/x711.asn1`, which is a formalization of the X.711 standard.

As an example of filtering, to find all of the `OMNIPoint` logs under the system object, any of the expressions shown below would be a valid argument to `Album::set_derivation()`.

```
/systemId="mysys"/LV(1)/CMISFilter(item:equality:{objectClass,log})
 or
LV(1)/CMISFilter(item: equality: {objectClass, log})
 or
*/CMISFilter(item:equality:{objectClass,log})
```

Building on the previous examples, to get only logs that are enabled, filter on `operationalState`, using `and`, as shown below:

```
LV(1)/CMISFilter(
     and: {
     item: equality: {objectClass, log},
     item: equality: {operationalState, enabled}
)
```

For the PMI, newlines are allowed and might enhance readability. Note that newlines might not be accepted by a shell.

3.3.7.4 Operation of a Filtering Derivation

When a filtered album is derived, the filtering is done automatically by the platform, so you never see any callbacks for the objects that are bypassed by the filter. If the album's `TRACKMODE` is set to `TRACK`, the album is maintained on the basis of the filter. That is, if an attribute changes in a way that makes the value of the filter true (when it has been false) or false (when it has been true), the album is automatically updated so that the image of the object is included (or excluded, as appropriate).

If the album is not set to `TRACK`, you can perform another scoped and filtered `M-GET` by calling `derive()` again.

If you execute `all_destroy()` on an album that has a derivation and that album is in a `DOWN` state, or is an `AUTOIMAGE` album, then the request is optimized to do a single scoped `M-DELETE` using the scope and filter specified for the album. (Otherwise a separate `M-DELETE` is issued for each member of the album, as before.)

The other `ALL` operations are not yet optimized in this way, but you can get the effect of an optimized `all_boot()`. If you execute `derive()` on an `AUTOIMAGE` album, the initial scoped `M-GET` fetches all the attributes at that time, rather than issuing a subsequent `M-GET` for each image.

3.4 Meta Data Repository

The Meta Data Repository (MDR) is a storage area within the MIS for the descriptions of managed objects. A description for every object known to the MIS is stored in the MDR. This data encompasses everything from the syntax required to refer to the attribute, to the composition of an object package. The MDR is initialized and updated by using the GDMO and ASN.1 compilers. MDR supports many actions to provide information about the objects.

3.4.1 `getAttribute` *Action*

```
getAttribute '"GDMO DOCNAME":attrName'
```

This action provides information about an attribute. Gives information about which ASN.1 module the attribute is actually defined.

3.4.2 `getAllDocuments` *Action*

```
getAllDocuments getAllDocuments
```

This action provides a list of all documents. You still have to provide either a class name or an oid as the argument.

3.4.3 `getAsn1Module` *Action*

```
getAsn1Module '"ASN1 DOCNAME"'
```

This action provides the complete information about an attribute in textual form. You can provide either the ASN.1 module name or the oid as the argument.

3.4.4 `getObjectClass` *Action*

```
getObjectClass '"GDMO DOCNAME":objectClass'
```

This action provides the complete information about a class in textual form, which includes all the class attributes and their properties. You can provide either the class name or the oid as the argument.

3.4.5 `getDocument` *Action*

```
getDocument '"GDMO DOCNAME"'
```

This action provides a list of all objects (and the oids) defined in a particular document. This action expects the document name.

3.4.6 `getOidName` *Action*

```
getOidName '{oid}'
```

For a given object identifier, returns the name of the object.

3.4.7 Sample Program

Following is a sample program to try out these different actions of the MDR.

Code Example 3-1 MDR Actions

```
#include <hi.hh>
#include <stdio.h>

Platform plat;
main(int argc, char **argv)
{
        char dn[1024];
        if (argc != 4) {
              printf("Usage dummy: mdr <hostname> < MDR action>
<arglist> \n");
                printf("\nSupported Actions:\n \n");
                printf("\t getObjectClass '\"GDMO
DOCNAME\":objectClass'\n");
                printf("\t getAllDocuments getAllDocuments\n");
                printf("\t getDocument '\"GDMO DOCNAME\"'\n");
                printf("\t getAttribute '\"GDMO
DOCNAME\":attrName'\n");
                printf("\t getAsn1Module '\"ASN1 DOCNAME\"'\n");
                printf("\t getOidName '{oid}'\n");
                printf("\nSample Usage commands:\n\n");
                printf("\t mdr host getAsn1Module '\"EM-TOPO-
ASN1\"'\n");
               printf("\t mdr host getOidName '{ 1 3 6 1 4 1 42
2 2 2 5 3 1
}'\n");
                printf("\t mdr host getAttribute '\"EM
TOPOLOGY\":topoNodeName'\n");
                printf("\t mdr host getDocument '\"EM
Topology\"'\n");
                printf("\t mdr host getObjectClass '\"EM
TOPOLOGY\":topoNode'\n");
```

Code Example 3-1 MDR Actions

```
#include <hi.hh>
                printf("\t mdr host getAllDocuments
getAllDocuments\n");
                exit(0);
        }
        char *host = argv[1];
        char *host = argv[1];
        char *action = argv[2];
        char *mod = argv[3];


        plat = Platform(duEM);
        printf("Connecting to %s ... ",host);
        plat.connect(host, "test_get");
        printf("Done.\n");
        printf(dn,"/systemId=\"%s\"/metaName=\"MDR\"", host);
        Image mdr = Image(dn);
        mdr.boot();
        Syntax in = mdr.get_param_syntax(action);
        Syntax res = mdr.get_result_syntax(action);
        printf("Input Syntax is %s\n",in.get().chp());
        printf("\n-------------------------------------\n");
        printf("Result Syntax is %s\n",res.get().chp());
        DU mdr_data = mdr.call(action, mod);
        printf("--> %s\n",mdr_data.chp());
}
```

3.5 Symbolic Constants and Defined Types

3.5.1 Constants

Following is an example of a symbolic constant:

```
const Timeout DEFAULT_TIMEOUT = -12345.0;
```

DEFAULT_TIMEOUT is the default argument to several of the member functions of the Platform class. -12345.0 is a distinguished value that causes those functions to substitute the value of the TIME_OUT property from the Platform instance. Refer to Section 3.5.2.6, “Timeout,” for more information. For example:

```
const DU duREPLACE = “REPLACE”;

const Callback NO_CALLBACK;

enum Platformid
     { VOID_PLATFORM_ID,
       G2_PLATFORM_ID,
     };
```

3.5.2 Defined Types

This section lists the defined types. They are declared in the /opt/SUNWconn/em/include/pmi/hi.hh file.

3.5.2.1 CCB

```
typedef const Callback& CCB;
```

3.5.2.2 CDU

```
typedef const DataUnit& CDU;
```

3.5.2.3 DU

```
typedef DataUnit DU;
```

3.5.2.4 FBits

```
typedef U32 FBits;
```

The bits have the following meanings on a `get`:

Table 3-4 Format Bit Values on `get` Function Calls

Format Bit	**Description**
`USE_NUMERIC_NAMES`	Do not translate OIDs to names
`OMIT_NEWLINES`	Do not “pretty-print”. Format only one line of output
`USE_C_ESCAPES`	Format control characters as C does: \n, \033, etc
`USE_EXPLICIT_TYPES`	Format type tags on ANY values
`OMIT_SPACES`	Omit all nonessential space characters
`USE_EXPLICIT_CHOICE`	Format choice with an explicit choice tag

The bits have the following meanings on a set:

Table 3-5 Format Bit Values on set Function Calls

Format Bit	Description
USE_NUMERIC_NAMES	(Ignored)
OMIT_NEWLINES	(Ignored)
USE_C_ESCAPES	Parse control characters as C does: \n, \033, etc.
USE_EXPLICIT_TYPES	Require type tags on ANY values
OMIT_SPACES	(Ignored)
USE_EXPLICIT_CHOICE	Expect choice to have an explicit choice tag.

3.5.2.5 FormatBits

```
Enum FormatBits
{ USE_NUMERIC_NAMES  = 1,
  OMIT_NEWLINES      = 2,
  USE_C_ESCAPES      = 4,
  USE_EXPLICIT_TYPES = 8,
  OMIT_SPACES        = 16,
};
```

3.5.2.6 Timeout

```
typedef double Timeout ;
```

3.6 Error Handling and Event Dispatching

3.6.1 Error Handling

Error handling is provided by the base class Error. Each of the object classes is derived from the Error class, except for the classes, AlbumImage.

3.6.2 Event Dispatching

The function `dispatch_recursive()` maintains queues of callback routines. One queue is maintained for each of the following: input, output, exception, and timers. These queues are scanned in the following order:

1. Exception
2. Output
3. Timer
4. Input

You associate a file descriptor with each callback on a queue when you use a function such as `post_fd_read_callback()`. When you call `dispatch_recursive()`, this function does a select on all the open file descriptors to determine their state, and then goes through each queue in the order indicated above to determine if there is outstanding data to be read from, or written to, the file descriptor.

Either FALSE or TRUE can be passed as a value to dispatcher as an argument. If FALSE is the value passed, then the select on the open file descriptors is done with a time-out value of 0. If TRUE is passed, then a short time interval is specified as the time-out.

For more information on using `dispatch_recursive()`, refer to the discussion of the `em_cmipconfig` example in the "Examples" chapter of the *Solstice Enterprise Manager Application Development Guide*.

3.7 High-Level PMI Classes

Each of the following classes is implemented as a reference-counting outer class wrapped around an inner abstract-base class. The PMI requires no access to the inner class.

Table 3-6 High-Level PMI Classes

Class	**Description**
Album Class	Represents a set of related objects
AlbumImage Class	Represents the state of an iterater
AppTarget Class	Represents target applications

Table 3-6 High-Level PMI Classes

Class	**Description**
AuthApps Class	Used when you need to know which applications a user is authorized to use
AuthFeatures Class	Used when you need to implement the feature level access control in your application
Coder Class	Represents a pair of methods for encoding and decoding values
CurrentEvent Class	Represents an event
Error Class	Stores details of errors related to an object instance
Image Class	Represents an actual or potential object in an MIS' framework
Morf Class	Represents a unit of data
PasswordTty Class	Implements the TTY based password query mechanism
Platform Class	Represents a potential or actual connect to a MIS
Syntax Class	Represents a type
Waiter Class	Represents an ongoing asynchronous operation

Note – As all class destructors have an identical format, i.e. ~<*class name*>(), the various class destructors are not specified in the following sections. Each class destructor, might or might not destroy the underlying <*class name*> object, depending on the reference count.

3.7.1 `Album` *Class*

Inheritance: `public Error`

```
#include <pmi/hi.hh>
```

Data Members: No public data members declared in this class

An album comprises a set of related objects, implemented as a set of `images`. Like mathematical sets, albums can be constructed either by rule or by enumeration. An album instance allows you to perform certain operations on each of the `images` that it contains. Like `images`, albums can be synchronized either manually or automatically.

The Album class allows an attribute list to be specified as an argument; in this way memory consumption of images is reduced. As examples, see `derive()` and start_derive`()`.

For more information on using the `Album` class, refer to the discussion of the `em_cmipconfig` example in the "Examples" chapter of the *Solstice Enterprise Manager Application Development Guide.*

Table 3-7 Album Method Types

Method Name	**Method Type**
`find_by_nickname`	Global lookup

`get_prop`
`set_prop`
`all_set_prop`
`all_set_attr_prop`
	Control properties

`get_derivation`
`set_derivation`
`derive`
`start_derive`
	Define membership of a derived album

`include`
`exclude`
`clear`
	Manipulate membership of an enumerated album

`num_images`	Census info
`get_userdata` `set_userdata`	User-defined properties
`first_image` `all`	Iterate over all images

`all_boot`
`all_start_boot`
`all_shutdown`
`all_start_shutdown`
	Image activation

Table 3-7 Album Method Types *(Continued)*

Method Name	**Method Type**

`all_set`
`all_set_long`
`all_set_str`
`all_set_gint`
`all_set_dbl`
`all_set_raw`
`all_revert`
`all_store`
`all_start_store`
`all_set_from_ref`
	Setting attributes

`all_create`
`all_start_create`
`all_create_within`
`all_start_create_within`
`all_destroy`
`all_start_destroy`
	Object existence

`all_call` `all_start`	Miscellaneous methods
`all_when`	Object events
`get_when_syntax` `when`	Album events

3.7.1.1 Constructors

Default Constructor

```
Album()
```

The default constructor creates an album instance that refers to no actual album object. The value tests false until you assign it a real album value.

Copy Constructor

```
Album(const Album& <other>);
```

This is an ordinary copy constructor. After the copy, both copies still refer to the same album object. The reference count on the album object is incremented.

Album Constructor

```
Album(CDU <nickname>,
     Platform& <p> = Platform::default_platform(), )
```

This constructor constructs an album instance for a particular kind of `Platform`. Because an album is really a wrapper for a set of related classes, this function actually works a bit like a virtual constructor.

3.7.1.2 Operator Overloading for `Album`

Assignment Operator

```
Album& operator = (constr Album& <other>)
```

The assignment operator works just like the copy constructor.

Cast Operator

```
operator void *();
```

The cast operator is for use in conditionals. It returns true if this album refers to an actual album object. Do not attempt to use the returned value as a pointer to anything, since it points to private data.

Not Operator

```
operator !();
```

This is provided so that you can say “`if (!album)` …“

3.7.1.3 *Member Functions of* `Album`

This section describes the member functions of the `Album` class.

all

Purpose: Call another function on each of the `image`s in the album.

```
Result all(Result (*<f>)( Image &<im>, void* <data>),
     void* <data> = 0)
```

- The first argument *<f>* is a pointer to the subfunction.
- The second argument is some arbitrary data to pass to the subfunction along with each image.

The subfunction’s syntax (embedded in the declaration of `all`) is therefore required to be :

```
Result <f>(Image &<im>, void* <data>) ;
```

The implementation of `all` calls the subfunction repeatedly, supplying an appropriate value for <*im*> to refer in turn, to each of the album’s `image`s, and passing to it each time the arbitrary data that was passed to `all`.

The `all` function itself returns a true value if all of the calls to the subfunction designated by <*f*> return true. A thrown exception terminates the iteration.

_all_boot_

Purpose: Perform a `boot` on each of the images in the album.

Syntax

```
Result all_boot(Timeout <to> = DEFAULT_TIMEOUT)
```

The timeout value is reset on completion of each successful boot. This function returns a true value if all image operations succeeded.

Example: Boot all images in an album.

```
Album bunch = Album(“demoalbum”); // Define, construct bunch.
bunch.set.derivation( "/LV(2)" ); // Derive the album.
Timeout to ;
...
if ( !bunch.all_boot( to )) {     // Boot all images in bunch.
    cout << “Using all_boot(): boot of one image failed.” ;
    exit ( 1 ) ;
}
```

all_call

Purpose: Perform a `call` on each of the images in the album.

```
Result all_call(CDU <name>,
    CDU <param>,
    Timeout <to> = DEFAULT_TIMEOUT)
```

<*name*> is the name of the action.

<*param*> is the parameter associated with the action.

<*to*> is the timeout.

If <*param*> is not provided (or if it is set to `duNONE`, there is no parameter associated with this action. The time-out is reset on completion of each successful call. Returns a true value if all image operations succeeded.

Example: Call all images in an album.

```
Album bunch = Album(“demoalbum”); // Define, construct bunch.
bunch.set.derivation( "/LV(2)" ); // Derive the album.
CDU nm, prm ;
Timeout to ;
...
if (!bunch.all_call( nm, prm, to)) {  // Call all images in bunch.
    cout << “Using all_call(): call of one image failed.” ;
    exit ( 1 ) ;
}
```

all_create

Purpose: Perform a `create` on each image in the album.

```
Result all_create( Image &<refobj> = Image(),
    Timeout <to> = DEFAULT_TIMEOUT)
```

The `Timeout` value is reset on completion of each successful `create` call. Returns a true value if all image operations succeeded.

Example: Create an object for each image in an album.

```
Album bunch = Album(“demoalbum”); // Define, construct bunch.
bunch.set.derivation( "/LV(2)" ); // Derive the album.
Image robj;
Timeout to ;
// Set name and class before invoking bunch.all_create(). See create.cc
...
if ( !bunch.all_create(robj, to)) { // Create object of each image in bunch.
    cout << “Using all_create(): create of one object failed.” ;
    exit ( 1 ) ;
}
```

all_create_within

Purpose: Perform a `create_within` on each of the images in the album.

```
Result all_create_within(CDU <container_objname>,
    Image &<refobj> = Image(),
    Timeout <to> = DEFAULT_TIMEOUT)
```

The Timeout value is reset on completion of each successful `create_within` call. Returns a true value if all image operations succeeded.

Example: Create an object within a container, for each image in an album.

```
...
if ( !bunch.all_create_within(cobj, robj, to)) { // Create object of each image in bunch.
    cout << “Using all_create_within(): create of one object failed.” ;
    exit ( 1 ) ;
}
```

all_destroy

Purpose: Perform a `destroy` on each of the images in the album.

```
Result all_destroy(Timeout <to> = DEFAULT_TIMEOUT)
```

The Timeout value is reset on completion of each successful `destroy` call. Returns a true value if all image operations succeeded.

Example: Destroy each image in an album.

```
...
if ( !bunch.all_destroy(to)) { // Create object of each image in bunch.
    cout << “Using all_destroy(): create of one object failed.” ;
    exit ( 1 ) ;
}
```

all_revert

Purpose: Perform a `revert` on each of the images in the album.

```
Result all_revert()
```

Returns a true value if all image operations succeeded.

all_set

Purpose: Perform a `set` on each of the `images` in the album.

```
Result all_set(CDU <name>,
    CDU <value>,
    CDU <op> = duREPLACE)
```

Returns a true value if all image operations succeeded.

all_set_attr_prop

Purpose: Set a property of an attribute in each of the `images` of current MIS.

```
Result all_set_attr_prop(CDU <name>,
     CDU <key>,
     CDU <value>)
```

This function sets a property for some attribute in each of the `images` of the current MIS, provided that:

- <*name*> specifies an existing attribute in the object class
- <*key*> specifies a supported property
- <*value*> specifies a legal value.

If `all_set_attr_prop` cannot do what you ask, it throws the Invalid exception.

Refer to `get_attr_prop`, under the description of `Image`, which starts on page 3-59, for some typical properties. Returns a true value if all image operations succeeded.

all_set_dbl

Purpose: Perform a `set_dbl` on each of the `images` in the album.

```
Result all_set_dbl(CDU <name>,
     double <value>,
     CDU <op> = duREPLACE)
```

Returns a true value if all image operations succeeded.

all_set_from_ref

Purpose: Perform a `set_from_ref` on each of the `images` in the album.

```
Result all_set_from_ref( Image& <refobj>)
```

Returns a true value if all image operations succeeded.

all_set_gint

Purpose: Perform a `set_gint` on each of the `images` in the album.

```
Result all_set_gint(CDU <name>,
     GenInt& <value>,
     CDU <op> = duREPLACE)
```

Returns a true value if all image operations succeeded.

all_set_long

Purpose: Perform a `set_long` on each of the `images` in the album.

```
Result all_set_long(CDU <name>,
     long <value>,
     CDU <op> = duREPLACE)
```

Returns a true value if all image operations succeeded.

all_set_prop

Purpose: Set a property for each of the `images` of the current album.

This function sets a property for each of the `images` of the current album, provided that:

- <*key*> specifies a supported property.
- <*value*> specifies a legal value.

If `all_set_prop` cannot do what you ask, it throws the Invalid exception.

```
Result all_set_prop(CDU <key>, CDU <value>)
```

Refer to `get_prop`, under the description of `Image`, for some typical properties. Returns a true value if all image operations succeeded.

all_set_raw

Purpose: Perform a `set_raw` on each of the `images` in the album. Returns a true value if all image operations succeeded.

```
Result all_set_raw(CDU <name>,
    Morf& <value>,
    CDU <op> = duREPLACE)
```

all_set_str

Purpose: Perform a `set_str` on each of the `images` in the album.

Returns a true value if all image operations succeeded.

```
Result all_set_str(CDU <name>,
     CDU <value>,
     CDU <op> = duREPLACE,
     FBits <fb> = 0)
```

The difference between `set_str` and `set` is that the data language used by `set` requires quotes as part of the string, while `set_str` assumes them if necessary. They are not always necessary; you can also pass numeric values as strings, and they are converted for you.

Refer to the discussion on MOD_PENDING in Table 3-13 on page 3-73 for a description of legal operations. For a list of possible <*fb*> values, refer to Section 3.5.2.5, “FormatBits,” on page 3-15.

all_shutdown

Purpose: Perform a shutdown on each of the images in the album.

The Timeout value is reset on completion of each successful shutdown. Returns a true value if all image shutdowns succeed.

```
Result all_shutdown(Timeout <to> = DEFAULT_TIMEOUT)
```

all_start

Purpose: Perform an asynchronous version of all_call.

The callback is called only once when all images are complete.

```
Waiter all_start(CDU <name>,
     CDU <param>,
     CCB <cb> = NO_CALLBACK)
```

all_start_boot

Purpose: Perform an asynchronous version of all_boot.

The callback is called only once, when all images are complete.

```
Waiter all_start_boot(CCB <cb> = NO_CALLBACK)
```

all_start_create

Purpose: Perform an asynchronous version of all_create.

The callback is called only once, when all images are complete.

```
Waiter all_start_create( Image& <refobj> = Image(),
     CCB <cb> = NO_CALLBACK)
```

```
Waiter all_start_create_within(CDU <container_objname>,
     Image &<refobj> = Image(),
     CCB <cb> = NO_CALLBACK)
```

The preceding function call is the asynchronous version of `all_create_within`. The callback is called only once, when all `images` are complete.

all_start_destroy

```
Waiter all_start_destroy (CCB <cb> = NO_CALLBACK)
```

The preceding function call is the asynchronous version of `all_destroy`. The callback is called only once, after all `images` are destroyed.

all_start_shutdown

```
Waiter all_start_shutdown(CCB <cb> = NO_CALLBACK)
```

The preceding function call is the asynchronous version of `all_shutdown`. The callback is called only once, after all image shutdowns are complete.

all_start_store

```
Waiter all_start_store(CCB <cb> = NO_CALLBACK)
```

The preceding function call is the asynchronous version of `all_store`. The callback is called only once, after all image stores are complete.

all_store

```
Result all_store(Timeout <to> = DEFAULT_TIMEOUT)
```

The preceding function call performs a `store` on each of the `images` in the album. The Timeout value is reset on completion of each successful store. It returns a true value if all image store succeeds.

all_when

```
Result all_when(CDU <eventname>,
                 CCB <cb> = NO_CALLBACK)
```

The preceding function call performs a `when` on each of the `images` in the album. For a list of supported events, refer to `when`, under the description of `Image`. Returns a true value if all image `when` operations succeed.

clear

```
Result clear()
```

The preceding function call nulls out this album, which is presumably of the enumerated variety. (The `clear` function is not as useful with derived albums, since such albums either track membership automatically or are repopulated by calling `Album::derive` again.)

derive

```
Result derive(Timeout <to> = DEFAULT_TIMEOUT)
```

The preceding function call causes the album membership list to be computed (or recomputed) using the derivation specified in the property `DERIVATION`. All previous membership information is lost. You can initialize a non-tracking album with a `derive` and then maintain it with `include` and `exclude`. A tracking album does not track until the first `derive` is done, so it works much like a `boot` does on an image.

derive

```
virtual Result derive (Array (DU) <attrlist>,
Timeout <to> = DEFAULT_TIMEOUT)
```

This function call allows a user to supply a list of attributes to be tracked when deriving an auto-imaging album. Attributes that are specified in the list are automatically tracked in each of the images in the derived album.

exclude

```
Result exclude(Image& <im>)
```

The preceding function call deletes from this album the image <*im*>.

```
Result exclude(Album& *<ai>)
```

The preceding function call deletes from this album the set of `image`s that are in the album <*ai*>.

Note – Inclusion and exclusion are simple procedural operations for use on enumerated albums, and do not "define" the membership of the album in any way that would override subsequent membership operations.

find_by_nickname

```
static Album find_by_nickname(CDU <name>,
     Platform& <plat> = Platform::def_platform))
```

The preceding function call looks up an album by its nickname and returns a pointer to the corresponding instance of album. Since albums don't generally have object names, this is the only way to find an album by name.

first_image

```
AlbumImage first_image()
```

The preceding function call returns the first image in the album, in the form of an `AlbumImage` iterater. The `AlbumImage::next_image` function returns subsequent `images` until there are no more.

Note – If you are tempted to use this iterater to see whether an image is a member of an album, remember that the `Image::is_in_album` function does this much more efficiently.

---

get_derivation

```
DU get_derivation()
```

The preceding function call is a shortcut function for the `get_prop` function.

get_prop

```
get_prop("DERIVATION") ;
```

```
Du get_prop(CDU <key>)
```

The preceding function call returns a property of the current album. If the *<key>* does not specify an existing property, a null `DataUnit` is returned, which tests false in a conditional.

Most albums support at least the properties described in Table 3-8.

Table 3-8 Properties Supported by Most Albums

Properties	**Description**
`STATE`	Whether the album is actively tracking: DOWN - The album is not tracking. BOOT - The album is currently being derived. UP - The album is tracking. SHUTDOWN - The album is shutting down.
`DERIVATION`	How this album relates to other albums and images by scoping and filtering. Refer to Section 3.3.7, “Filtering as an Aspect of Album Derivation,” on page 3-6.
`NICKNAME`	The nickname of the album.
`TRACKMODE`	How the membership of the album is to be maintained: SNAP - the album does not maintain its membership list, but relies on explicit `derive()`, `include()`, and `exclude()` calls to tell the album which images are to be included. Note that a SNAP album might contain tracking images. TRACK — the album analyzes its derivation rule and automatically includes or excludes images as they change, when they match or fail to match the derivation rule. Refer to the `AUTOIMAGE` property if you want included images to boot and start tracking automatically.

`AUTOIMAGE`
`(YES, NO)`
	For any image included in the album, also automatically boots the image as a tracking image. (This happens before any of the registered callbacks, if any, are called.)

`ACCESS (RWRWRW)`	The access permissions for this album, if persistent.

get_userdata

```
DU get_userdata(CDU <key>)
```

The preceding function call returns any data stored by the application under the <*key*> specified. There are no predefined values. If there is no data under that <*key*> for this instance, the return value (a null `DataUnit`) evaluates to false.

get_when_syntax

```
Syntax get_when_syntax(CDU <eventtype>)
```

The preceding function call returns the `Syntax` of a given event type. The event information comes into the callback as a `CurrentEvent`, from which the event information can be extracted.

include

```
Result include(Image& <im>)
```

The preceding function call adds the specified image to this album.

```
Result include(Album& <al>)
```

The preceding function call adds the set of images in the album *al* to this album.

Note – Inclusion and exclusion are simple procedural operations for use on enumerated albums, and do not "define" the membership of the album in any way that would override subsequent membership operations.

---

num_images

```
U32 num_images()
```

The preceding function call returns the number of `images` in this album.

set_derivation

```
Result set_derivation(CDU <derivation>)
```

The preceding function call is a shortcut function for the following.

```
set_prop("DERIVATION", <derivation>) ;
```

```
Result: set_prop(CDU <key>, CDU <value>)
```

The preceding function call sets a property of the current album, provided that <*key*> specifies a supported property and <*value*> specifies a legal value. If `set_prop` cannot do what you ask, it throws the Invalid exception. Refer to `get_prop` under the description of the `Album` class, which starts on page 3-17, for some typical properties.

set_userdata

```
Result set_userdata(CDU <key>, CDU <value>)
```

The preceding function call stores arbitrary data supplied by the application under the <*key*> specified. There are no predefined values. If this instance already has data under that <*key*>, the new data replaces the old without comment. Essentially, user data is an associative array belonging to the album; an application can use it any way it pleases.

start_derive

```
Waiter start_derive(CCB <cb> = NO_CALLBACK)
```

The preceding function call is the asynchronous version of `derive`.

```
virtual Waiter start_derive( Array (DU) <attrlist>,
CCB <cb> = NO_CALLBACK)
```

This function call is the asynchronous version of the `derive` function that accepts a list of attributes.

when

```
Result when(CDU <eventname>,
     CCB <cb> = NO_CALLBACK)
```

The `Platform` object receives all events at which time all callbacks registered for by the `Platform` objects are executed. Next, all of the callbacks registered for by image objects are executed, then all of the callbacks registered for by album objects are executed.

The preceding function call establishes a callback routine to handle an album-specific asynchronous event.

For example, you might want to know if any object was destroyed. You could say:

```
when("OBJECT_DESTROYED",Callback(destroyed_cb, 0)) ;
```

The following `Album` events are supported:

Table 3-9 Events Supported by Album

Events	**Description**
`IMAGE_INCLUDED`	An image was added to the album by some means. The new image is not automatically booted unless the `AUTOIMAGE` property was set. The `CurrentEvent::do_something()` function does nothing. This is an internal event, and has no MIS event info associated with it. Note: Use the method `Image::exists()` to see whether a given image from within `IMAGE_INCLUDED` exists.
`IMAGE_EXCLUDED`	An image was deleted from the album by some means or other. The `CurrentEvent::do_something()` function does nothing. Note that this is an internal event, and has no MIS event info associated with it.

Table 3-9 Events Supported by Album

`OBJECT_CREATED`	An object in the album came into existence. Call `CurrentEvent::do_something()` to cause inclusion in a tracking album before the end of the callback.
`OBJECT_DESTROYED`	An object in the album was destroyed. Call `CurrentEvent::do_something()` to cause exclusion in a tracking album before the end of the callback.
`RAW_EVENT`	Some album-related event occurred. You can examine it before the PMI does anything with it. The `CurrentEvent::do_something()` function does nothing.

Refer to the description of the `CurrentEvent::do_something` and `CurrentEvent::do_nothing` member functions in Section 3.7.7.3, "Member Functions of CurrentEvent," on page 3-48 for more information on these `CurrentEvent` member functions.

3.7.2 `AlbumImage` *Class*

Inheritance: `class AlbumImage`

```
#include <pmi/hi.hh>
```

Data Members: No public data members declared in this class

An instance of the `AlbumImage` class is an iterator that represents the current album in a list of albums or the current image in a list of `images`. It can also be viewed as a two-way association between a given image and a given album.

3.7.2.1 Constructors

Default Constructor

```
AlbumImage()
```

The default constructor creates an `AlbumImage` instance that refers to no actual `AlbumImage`. The value tests false until you assign it a real `AlbumImage` value.

Default Constructor

```
AlbumImage( AlbumImage& <other>)
```

This is an ordinary copy constructor. After the copy, both copies still refer to the same `AlbumImage` object. The reference count on the `AlbumImage` object is incremented.

3.7.2.2 Operator Overloading for `AlbumImage`

Assignment Operator

```
AlbumImage& operator = (const AlbumImage& <other>)
```

The assignment operator works just like the copy constructor.

Cast Operator

```
operator void *()
```

The cast operator is for use in conditionals. It returns true if this `AlbumImage` refers to an actual `AlbumImage` object. Do not attempt to use the returned value as a pointer to anything, since it points to private data.

Not Operator

```
int operator !()
```

This operator definition is provided so that you can say "`if (!albumimage)`..."

Album Operator

```
operator Album()
```

This returns the album pointed to by the current `AlbumImage` association.

Image Operator

```
operator Image()
```

This returns the image pointed to by the current `AlbumImage` association.

3.7.2.3 Member Functions of `AlbumImage`

This section describes the member functions of the `AlbumImage` class.

next_album

Purpose: Return the next album.

This is used when iterating over all of the albums of an image.

```
AlbumImage next_album()
```

next_image

Purpose: Return the next image.

This is used when iterating over all of the `images` of an album.

```
AlbumImage next_image()
```

For more information on using the AlbumImage class, refer to the section “Using the `AlbumImage` Class” in Chapter 4 of the *Solstice Enterprise Manager Application Development Guide*.

3.7.3 `AppTarget` *Class*

Inheritance: public `Album`

Data Members: No public data members declared in this class

3.7.3.1 Constructors

```
AppTarget::operator !()
```

AppTarget is an abstraction that represents target applications.

3.7.3.2 Operator Overloading for AppTarget

The ! is overloaded to check whether the Album is NULL or non-NULL.

3.7.4 `AuthApps` *Class*

Inheritance: class AuthPriv

#include <pmi/auth_apps.hh>

Data Members: No public data members declared in this class

This class is derived from `AuthPriv`, which is an abstract base class for this class as well as the `AuthFeatures` class. `AuthPriv` is not documented since you don't need to use it directly.

This class is used when you need to know which applications a user is authorized to use. Currently, this is used by the Launcher application to gray out the application icons which the user is not authorized to use.

After the application has successfully connected to the platform (using Platform::connect), an instance of this class is passed as an argument to the `Platform::get_authorized_apps` method. The method fills in the instance with the information about the applications the user is authorized to use. After the successful completion of this method, one can use the `AuthApps::is_authorized` method to find out whether an application is authorized or not. For more information, please refer to the description of the `Platform::get_authorized_apps` method.

3.7.4.1 *Constructors*

```
AuthApps ()
```

The default constructor, above, creates an empty instance of this class which can be passed as an argument to the `Platform::get_authorized_apps` method.

3.7.4.2 *Operator Overloading*

No public operators are defined for this class.

3.7.4.3 *Member Functions*

The following are member functions for the `AuthApps` class.

is_authorized

```
Result is_authorized (const char *appname) const
```

This is an inherited method from the `AuthPriv` class. This method returns OK if the user is authorized to access the given application. This method should be called after successful completion of the `Platform::get_authorized_apps` method. Otherwise, it will always return NOT_OK.

3.7.5 `AuthFeatures` *Class*

Inheritance: class AuthPriv

#include <pmi/auth_features.hh>

Data Members: No public data members declared in this class

This class is derived from `AuthPriv`, which is an abstract base class for this class as well as the `AuthApps` class. AuthPriv is not documented since you don't need to use it directly.

This class is used when you need to implement the feature level access control in your application. The application writer decides which features the application should have and what they control. The user who is running the application may not have access to all features. The application should query the list of features the user is authorized to use and accordingly restrict the operations the user can perform using the application.

After the application has successfully connected to the platform (using `Platform::connect`), an instance of this class is passed as an argument to the `Platform::get_authorized_features` method. The method fills in the instance with the information about the features the user is authorized to use. After the successful completion of the this method, one can use the `AuthFeatures::is_authorized` method to find out whether a feature is authorized or not. For more information, please refer to the description of the `Platform::get_authorized_features` method. For an example program, please refer to $EM_HOME/src/access_sample/access_feature_level.cc.

3.7.5.1 *Constructor*

```
AuthFeatures ()
```

The default constructor, above, creates an empty instance of this class which can be passed as an argument to the `Platform::get_authorized_features` method.

3.7.5.2 Operator Overloading

No public operators are defined for this class

3.7.5.3 Member Functions

The following are member functions for the `AuthFeatures` class.

is_authorized

```
Result is_authorized (const char *feature) const
```

This is an inherited method from the `AuthPriv` class. This method returns OK if the user is authorized to access the given feature. This method should be called after successful completion of the `Platform::get_authorized_features` method. Otherwise, it will always return NOT_OK.

3.7.6 `Coder` *Class*

Inheritance: `public Error`

```
#include <pmi/hi.hh>
```

Data Members: No public data members declared in this class

An instance of the `Coder` class specifies custom encoding and decoding functions for getting and setting the string value of a `Morf` or attribute. In general you would not call the methods of this class yourself. The PMI calls them when doing translation for various `get_str` and `set_str` operations. You set up these translation functions by deriving from the inner `CoderData` class, supplying virtual functions to do the appropriate translation, along with a virtual destructor to correctly destroy your `CoderData`. You can also declare other items in your derived class which are available to your routines.

The base CoderData class supplies an operator Coder to construct a Coder from a CoderData. Hence, the correct C++ incantation for creating a Coder is:

```
Coder(*new OiCoderData(<arg1>, <arg2>...));
```

You then register the Coder with either of Platform::set_attr_coder or Syntax::set_coder.

3.7.6.1 Constructors

Default Constructor

```
Coder()
```

The default constructor creates a Coder instance that refers to no actual Coder. The value tests false until you assign it a real Coder value.

Copy Constructor

```
Coder(const Coder& <other>)
```

This is an ordinary copy constructor. After the copy, both copies still refer to the same Coder object. The reference count on the Coder object is incremented.

3.7.6.2 Operator Overloading for Coder

Assignment Operator

```
Coder& operator = (const Coder& <other>)
```

The assignment operator works just like the copy constructor.

Cast Operator

```
operator void*()
```

This cast operator is for use in conditionals. It returns true if this `Coder` refers to an actual `Coder` object. Do not attempt to use the returned value as a pointer to anything, since it points to private data.

Not Operator

```
int operator !()
```

This operator definition is provided for conditionals such as:

```
    if ( !albumimage ) …
```

3.7.6.3 Member Functions of Coder

This section describes the member functions of the `Coder` class.

get_str()

Purpose: Translate a `Morf`'s value into a textual `DU` value.

```
DU get_str( Morf& <mf>, FBits <fb> )
```

set_str( )

Purpose: Translate a textual `DU` value into a `Morf`'s value.

```
Morf& set_str( Morf& <mf>, CDU <data>, FBits <fb> )
```

3.7.7 CurrentEvent Class

Inheritance: public Error

```
#include <pmi/hi.hh>
```

Data Members: No public data members declared in this class

An instance of the CurrentEvent class represents all the information that is known about an asynchronous event, available in a form that doesn't require the arbitrarily dangerous casting of various pointer values. A CurrentEvent is passed into every callback function, and is also returned by the Waiter::wait function. Within a callback, control is available both before and after any action that the PMI itself would perform on your behalf.

Table 3-10 CurrentEvent Method Types

Method Name	**Method Type**
do_nothing do_something handled	Control PMI performance
get_event get_event_raw get_info get_info_raw get_message get_name get_oid	Extract event information
get_album get_eventtype get_image get_objclass get_objname get_platform get_time	Extract contextual information

Table 3-10 CurrentEvent Method Types

Method Name	**Method Type**
`something_to_do`	Set control information

`set_event_raw`
`set_info_raw`
`set_message`
`set_name`
`set_oid`
	Set event information (called primarily by the PMI)

`set_album`
`set_eventtype`
`set_image`
`set_objclass`
`set_objname`
`set_time`
	Set contextual information (called primarily by the PMI)

3.7.7.1 *Constructors of* `CurrentEvent`

Default Constructor

```
CurrentEvent()
```

The default constructor creates a `CurrentEvent` instance that refers to no actual `CurrentEvent`. The value tests false until you assign it a real `CurrentEvent` value.

Copy Constructor

```
CurrentEvent( const CurrentEvent& <other>)
```

This is an ordinary copy constructor. After the copy, both copies still refer to the same `CurrentEvent` object. The reference count on the `CurrentEvent` object is incremented.

Calldata Constructor

```
CurrentEvent( Ptr <calldata>)
```

This constructs a `CurrentEvent` from the *<calldata>* pointer passed into a callback as its second argument.

3.7.7.2 *Operator Overloading for* `CurrentEvent`

Assignment Operator

```
CurrentEvent& operator = ( const CurrentEvent& <other> )
```

This assignment operator works just like the copy constructor.

Cast Operator

```
operator void*()
```

This cast operator is for use in conditionals. It returns true if this `CurrentEvent` refers to an actual `CurrentEvent` object. Do not attempt to use the returned value as a pointer to anything, since it points to private data.

Not Operator

```
int operator !()
```

This operator definition is provided for conditionals such as:

```
    if ( !cur_event ) …
```

3.7.7.3 *Member Functions of* `CurrentEvent`

This section describes the member functions of the `CurrentEvent` class.

do_nothing()

Purpose: Throw away the current event so that the PMI does nothing.

The PMI does not perform the action it would otherwise perform. This is meaningful only within a callback. If multiple objects receive callbacks for a given event (for example, one callback for an image, and one for the album containing the image), then each callback needs to call this function to disable the operations ordinarily done by the corresponding object.

```
void do_nothing()
```

Refer to Table 3-9 on page 3-36 for information on how the member function `CurrentEvent::do_nothing` applies to the `Album::when` member function.

do_something()

Purpose: Perform the customary action for this event.

This is meaningful only within a callback. Before a callback is executed in response to an event, the PMI figures out what it wants to do and queues that operation using the `something_to_do` function. At the end of the callback, if you have not handled the event explicitly by calling either `do_something` or `do_nothing`, the PMI calls `do_something` on your behalf.

```
void do_something()
```

Refer to Table 3-9 on page 3-36 for information on how the member function `CurrentEvent::do_something` applies to the `Album::when` member function.

get_album()

Purpose: Return the associated album, if any.

```
Album get_album()
```

get_event()

Purpose: Return, in *textual* form, the entire event message, if one exists.

```
DU get_event()
```

get_event_raw()

Purpose: Return, in *encoded* form, the entire event message, if one exists.

```
Morf get_event_raw()
```

get_eventtype()

Purpose: Return the type of event as specified by the `when` function that established the current callback.

```
DU get_eventtype()
```

get_image()

Purpose: Return the image associated with this event, if one exists.

```
Image get_image(BOOLEAN create = FALSE)
```

get_info()

Purpose: Return, in *textual* form, the central event information.

```
DU get_info()
```

get_info_raw()

The preceding function call returns in encoded form the central event information.

```
Morf get_info_raw()
```

get_message()

Purpose: Return a pointer to the event message.

This customarily returns a pointer to the event message, if one exists, though it could be used for any arbitrary data, depending on the event. You wouldn't generally need to call this function unless you were in some fashion cheating on the PMI. The returned pointer is unlikely to be meaningful outside the scope of the `CurrentEvent`, but if you're already cheating, you probably know what to do about that.

```
Ptr get_message()
```

get_name()

Purpose: Return the name of the event.

This is the actual event name, which is not necessarily the same name as the one you used in the `when` function call that caused the callback to be invoked. There is no event named `RAW_EVENT`, for instance. That information can be retrieved using `get_eventtype`, however.

```
DU get_name()
```

get_objclass()

Purpose: Return the name of the class of the associated image, if it exists.

```
DU get_objclass()
```

get_objname()

Purpose: Return the name of the associated image, if it exists.

```
DU get_objname()
```

get_oid()

Purpose: Return the OID.

This returns the OID corresponding to the event name, if one exists. Internal events like `IMAGE_INCLUDED` have no OID.

```
Oid get_oid()
```

get_platform()

Purpose: Return the platform.

This returns the MIS platform associated with this event.

```
Platform get_platform()
```

get_time()

Purpose: Return, in ISO format, the time of the event, if available.

```
DU get_time()
```

handled()

Purpose: Determine if the `CurrentEvent` was handled.

This returns true if and only if either of `do_something` or `do_nothing` has been called on the `CurrentEvent` for the current callback.

```
Boolean handled()
```

set_album()

Purpose: Set the associated album.

```
void set_album( Album& <al> )
```

set_event_raw()

Purpose: Set the encoded event message.

```
void set_event_raw( Morf& <mf> )
```

set_eventtype()

Purpose: Set the event type.

```
void set_eventtype( CDU <eventtype> )
```

set_image()

Purpose: Set the associated image.

```
void set_image( const Image& <im> )
```

set_info_raw()

Purpose: Set the central event information.

```
void set_info_raw( const Morf& <mf> )
```

set_message()

Purpose: Set the message pointer.

```
void set_message( Ptr<msg> )
```

set_name()

Purpose: Set the event name.

```
void set_album( CDU<nm> )
```

set_objclass ( )

Purpose: Set the object class.

```
void set_album( CDU <cl> )
```

set_objname()

Purpose: Set the object name.

```
void set_album( CDU <nm> )
```

set_oid()

Purpose: Set the event OID.

```
void set_album( Oid& <o> )
```

set_time()

Purpose: Set the time of the event.

```
void set_album( CDU <tm> )
```

something_to_do ()

Purpose: Queue up something to be done when `do_something` is called.

Ordinarily it is called by the PMI before a callback is called, but you could use it to queue up additional operations to occur after the PMI calls `do_something`. It's usually easier, however, to call `do_something` yourself within the callback and then do whatever else needs doing.

```
void something_to_do ( CCB <cb>, Ptr <cdata> )
```

3.7.8 `Error` *Class*

Inheritance: `class Error`

`#include <pmi/error.hh>`

The class `Error` stores details of errors related to an object instance of a derived-from-`Error` class, such as `Image`.

Example: If `im1` is declared and used as follows:

```
Image im1 ;
im1.f1() ;
```

...then if `f1()` fails, the error string and type can be retrieved with:

```
im1.get_error_string() ;
im1.get_error_type() ;
```

For any function `f1()` of a class such as `Image`, if `f1()` is a static function, then no object is associated with that function. In such a case, you can use the variable `error` for error handling, where `error` is declared as:

```
Error error ;
```

Example: `f2()` is a static function, in the class `Image`, called as follows:

```
Image::f2() ;
```

If `f2()` fails you can query errors as follows:

```
error.get_error_type() ;
error.get_error_string() ;
```

Refer to the example of error handling in the sample program `get.cc` in the “Examples” chapter in the *Solstice Enterprise Manager Application Development Guide*.

3.7.8.1 Constructor

The syntax for the constructor for the `Error` class is:

```
Error(ErrorType etype=PMI_SUCCESS)
```

3.7.8.2 Operator Overloading of `Error`

There are no overloaded operators in the `Error` class.

3.7.8.3 Public Data Member

The `Error` class has the following public data member:

```
static void (*error_entry_callback)(Error *)
```

The pointed-to callback function, if set, is called after entering a function of a class that is derived from `Error`. If that object instance is already in error, it might be because a previous function call had failed.

Refer to the example of error handling in the sample program `get.cc` in the "Examples" chapter in the *Solstice Enterprise Manager Application Development Guide*.

3.7.8.4 Member Functions of `Error`

This section describes the member functions of the `Error` class.

error_to_string

Purpose: Return the default error string for a type.

```
static const char *error_to_string( ErrorType etype);
```

get_error_string

```
char   *get_error_string(void) const
```

get_error_type

```
ErrorType get_error_type(void) const
```

set_error_string

```
void   set_error_string(char *)
```

set_error_type

```
void   set_error_type(ErrorType)
```

set_error

```
void   set_error(ErrorType,char *)
```

reset_error

```
void   reset_error(void)
```

set_error_entry_callback

```
static  void   set_error_entry_callback(void (*eec)(Error *)=0)
```

3.7.8.5 Error Types and Strings

Table 3-11 shows the error types and strings returned by `get_error_type()`

and `get_error_string()`.

Table 3-11 Error Types

Type	Comment
MIS_ACCESS_NO_CONNECT_PRIVILEGE	Missing application connect privilege
MIS_ACCESS_USER_DOES_NOT_EXIST	Missing user profile
MIS_ACCESS_USER_NOT_MEMBER_OF_ANY_GROUP	User is not a member of any access control group
MIS_APP_INST_CREATE_FAILED	Failed to create application instance
MIS_CONNECTION_PDU_PARSING_FAILED	Failure during ape connect
MIS_CREATE_CALLBACK_FAILED	Failure of ape instance to create callback
MIS_ERROR	Error in MIS (unexpected?)
MIS_INVALID_PASSWORD	Invalid password on MIS host
MIS_RESOURCE_LIMIT	Ran out of memory
MIS_USER_DOES_NOT_EXIST	User does not exist on MIS host
PMI_CONNECTION_REPLY_PARSING_FAILED	Data passed by an application is not in an expected format
`PMI_DATA_OBJECT_OP_FAILURE`	Operation on associated data object failed
PMI_EM_LOGIN_DEAMON_PROBLEM	Check em_login daemon on MIS host
`PMI_ENCODE_FAILED`	Encoding of attribute failed
`PMI_ERROR`	Error
`PMI_ILLEGAL_OPERATION`	This operation cannot be performed on attribute
`PMI_INVALID_ARGUMENT`	Argument is not valid in this context
`PMI_MESSAGE_SENDS_FAILED`	Sending of message failed
`PMI_NEW_FAILED`	New memory allocation failed
`PMI_NO_DATA_OBJECT`	Data object associated with this object is NULL
`PMI_NOT_IMPLEMENTED`	This feature is not implemented
`PMI_NULL_ARGUMENT`	Argument passed OR attribute used is NULL
`PMI_OPERATION_FAILED`	Some operation failed
`PMI_SUCCESS`	Success
`PMI_UNKNOWN_PLATFORM`	Platform is unrecognized
PMI_USER_ABORTED_CONNECTION	Canceled password dialog

3.7.9 `Image` *Class*

Inheritance: public Error

```
#include <pmi/hi.hh>
```

Data Members: No public data members declared in this class

An image is the local representation of a potential or actual framework object. If it represents an actual object, the image is capable of either manual or automatic synchronization. In many respects you can think of the image as the object itself, even though the actual object is off in the MIS, or even further away. Images give access to the object's methods and attributes.

For information on how to use the Image class, refer to the discussion of the `em_cmipconfig` example in the "Examples" chapter of the *Solstice Enterprise Manager Application Development Guide* (this book is provided on AnswerBook only).

The image provides a model in which data is primarily textual, but also allows raw data to be passed in the form of `Morf`s. Images also provide attribute-like access to object and attribute schema information.

Table 3-12 Image Method Types

Member Function	**Method Type**

`first_album`
`is_in_album`
`num_albums`
	Determine album membership

`find_by_nickname` `find_by_objname`	Global lookup

`boot`
`shutdown`
`start_boot`
`start_shutdown`
	`Image` Activation

`get_prop` `set_prop`	Control Properties
`get_userdata` `set_userdata`	User-defined information

Table 3-12 Image Method Types

Member Function	**Method Type**
set_nickname set_objclass	Set object information
exists get_objclass get_nickname get_objname get_state	Get object information
attr_changed attr_exists get_attr_names get_attr_prop get_attr_trackmode set_attr_prop	Attribute information
get_attr_numerrors get_attr_last_error	Get attribute error information
set set_dbl set_from_ref set_gint set_long set_raw set_str	Set imaginary attribute values
get get_dbl get_gint get_long get_raw get_str	Get attribute values since last attribute value change event notification from the managed object
get_set get_set_dbl get_set_gint get_set_long get_set_raw get_set_str	Get imaginary attribute values (before they've been stored)
revert start_store store	Realize or discard imaginary attribute values

Table 3-12 Image Method Types

Member Function	**Method Type**
`create` `start_create`	Object creation, known name
`create_within` `start_create_within`	Object creation, known container
`destroy` `start_destroy`	Object destruction

`call`
`call_raw`
`start`
`start_raw`
	Miscellaneous object method activation

`get_param_syntax` `get_result_syntax`	Method data formats
`get_when_syntax` `when`	Notification

3.7.9.1 *Constructors of* `Image`

In this section, be sure to reboot the object before performing operations on it.

Default Constructor

```
Image()
```

The default constructor creates an image instance that refers to no actual image object. The value tests false until you assign it a real image value.

Copy Constructor

```
Image(const Image& <im>)
```

This constructor is an ordinary copy constructor. After the copy, both copies still refer to the same image object.

General Constructor

```
Image ( CDU <objname> = CDU(),
    CDU <objclass> = CDU(),
    Platform& <plat> = Platform::default_platform(), )
```

This constructor creates an image instance for a particular kind of MIS. Because an image is really a wrapper for a set of related classes, this function actually works a bit like a virtual constructor.

If the object pointed to by <*objname*> already exists, the <*objclass*> parameter is not required. The <*objclass*> is populated in the object whenever a get / set / delete response is received from the MIS.

While creating the object, <*objclass*> is required, if the object does not exist, but such an object is creatable, then `boot()` succeeds. If the object is not creatable, then `boot()` fails. Refer to the description of `boot()` on page 3-64.

3.7.9.2 Operator Overloading for `Image`

Assignment Operator

```
Image& operator = (const Image& <other>)
```

The *assignment* operator, above, works just like the copy constructor.

Cast Operator

```
operator void*()
```

The *cast* operator, above, is for use in conditionals. It returns true if this image refers to an actual image object. Do not attempt to use the returned value as a pointer to anything, since it points to private data.

Not Operator

```
int operator !()
```

The *not* operator, above, is provided so that you can say “`if (!image)`…”

3.7.9.3 Member Functions of `Image`

This section describes the member functions of the `Image` class.

attr_changed

Purpose: Determine whether an attribute has changed.

It returns true if the *real* value of the named attribute was modified.

```
Boolean attr_changed( CDU <name> )
```

Calling Sequence

```
...
Image thing = Image(dn, clsnm);
CDU atnm = "abcXYZ" ; // Some valid attribute name for this class
...
if ( !thing.attr_changed( atnm ) ) {
    cout << "The attribute was modified.\n" ;
}
```

Note that this notification refers only to a change reported in the *last received* notification from the MIS that an attribute was changed. The notification might be either an expected response to a store request of your own, or a somewhat-less-expected attribute change notification caused by someone else’s store request. In the latter case, if you’ve asked for the `ATTR_CHANGED` notification, the image’s real attributes do not change until the `CurrentEvent::do_something` function is called, either by you within the callback, or by the PMI after your callback returns. This allows you to get at the attribute values both before and after the change takes effect.

attr_exists

Purpose: Determine whether an attribute exists.

It can return false on an existing attribute if the value of EXISTS at that point is MAYBE.

```
Boolean attr_exists( CDU <name>)
```

Calling Sequence

```
...
Image thing = Image(dn, clsnm);
CDU atnm = "abcXYZ" ; // Some valid attribute name for this class
...
if ( !thing.attr_exists( atnm ) ) {
    cout << "The attribute does not exist.\n" ;
}
```

boot

Purpose: Activate an image and determine whether the object exists.

```
Result boot( Array (DU) <attrlist>,
       Timeout <to> = DEFAULT_TIMEOUT)
```

If it does exist, any attributes not marked IGNORE become available for the various `get` functions. (If it does not exist, you can set attributes, then make a `create` or `create_within` function call.) If TRACKMODE was set to TRACK, the image also begins tracking changes to its object. When the boot is complete, the STATE of the image is set to UP.

Calling Sequence

```
...
Image im(fdn);
if( !im.boot() ) {  // Boot an image without an attribute list.
            printf("Error in boot\n");
            return 2;
}
```

For a complete example, refer to the `sample/image_boot.cc` file.

For an existing object, if the class specified at the time of object construction is incorrect, this is an error, and `boot()` fails. Refer to the description of the `Image` constructors on page 3-61.

call

Purpose: Call a method for an image, with parameter as textual data language.

Call a miscellaneous method for the managed object represented by this image. Both parameter and result are passed as textual data language.

```
DU call( CDU <name>,
         CDU <param> = duNONE,
         TIMEOUT <to> = DEFAULT_TIMEOUT)
```

In the example above, <*name*> is the name of the action. <*param*> is the parameter associated with the action. If <*param*> is not provided (or if it is set to duNONE), there is no parameter associated with this action specified in the GDMO.

The syntax of the parameter can be found using the `get_param_syntax` function. The syntax of the result is found using the `get_result_syntax` function. Action requests sent as a result of call are always sent with the confirmed bit set. To send unconfirmed action requests, please refer to `Image::start()`. This function (`Image::start()`) also sends unconfirmed action requests message when callback is NO_CALLBACK.

call_raw

Purpose: Call a method for an image, with parameter encoded.

Call a miscellaneous method for the managed object represented by this image. The parameter must be in encoded form.

```
Morf call_raw( CDU <name>,
               Morf <param>,
               TIMEOUT <to> = DEFAULT_TIMEOUT)
```

In the example above, <*name*> is the name of the action. <*param*> is the parameter associated with the action. If <*param*> is not provided (or if it is set to duNONE, there is no parameter associated with this action.

The syntax of the parameter can be found using `get_param_syntax`. The syntax of the result can be found using `get_result_syntax`. Action requests sent as a result of `call` are always sent with the confirmed bit set. To send unconfirmed action requests, please refer to `Image::start()`.

create

Purpose: Create a managed object represented by a given image.

```
Result create( Image& <refobj> = Image(),
               Timeout <to> = DEFAULT_TIMEOUT)
```

Calling Sequence

```
Image thing = Image(dn,oc);
    before the next call, thing must be a valid image for an object. See create.cc
if ( !thing.create() ) {
                cout << "create failed" << endl;
                return 1;
        } else {
                cout << "create succeeded" << endl;
        }
```

You cannot create an object that already exists; however, you can test before the create to see if it already exists.

For a complete example, refer to the `sample/create.cc` file.

The attributes of the new object are a combination of the attributes supplied in the imaginary object, plus any additional attributes supplied by the reference object (if one was supplied), plus any other attributes the MIS feels like creating.

The `create` function requires the following:

- The object class of the image is set correctly

- There is sufficient information in the object name, nickname, and/or attributes for the MIS to figure out the complete object name for the new object (to the extent that it can't simply make up a name).

After the `create`, the `OBJNAME` property has been set to the actual complete object name, even if it was not specified completely before the `create`.

create_within

Purpose: Create, an object in a container.

Within a specified container object, create, the managed object represented by the specified image.

```
Result create_within( CDU <objname>,
                      Image& <refobj> = Image(),
                      const Timeout <to> = DEFAULT_TIMEOUT)
```

For `create_within()`, first construct the image using the `duNone` constant as the instance name (instead of the fdn) as follows:

```
im = Image(duNone, object_class);
im.create_within(args ...);
```

Calling Sequence

```
...
Image thing = Image(duNone,oc);
if ( !thing.create_within() ) {
                cout << "create_within failed" << endl;
                return 1;
      }
```

Typically you would use this when you want the MIS to make up a name for your object, and you only want to specify the object’s location. You cannot create an object that already exists, or you receive an Invalid exception. The attributes of the new object are a combination of the attributes supplied in the imaginary object, plus any additional attributes supplied by the reference object (if one was supplied), plus any other attributes the MIS feels like creating.

The `create_within` function requires the following:

- The object class of the image be set correctly.
- There is sufficient information in the container object name plus the attributes for the MIS to figure out the complete object name for the new object (to the extent that it cannot simply make up a name).

After the `create`, the `OBJNAME` property is set to the actual complete object name, even though it wasn't specified before the `create`.

destroy

Purpose: Destroy the managed object represented by this image.

Depending on the semantics of the MIS at that point, this might also delete any children of the object in question, or it might refuse to delete anything if there are any children.

```
result destroy(Timeout <to> = DEFAULT_TIMEOUT)
```

Calling Sequence

```
...
Image thing = Image(dn,oc); // Valid image of an object.
. . .
if ( !thing.destroy() ) {
           cout << "destroy failed" << endl;
           return 1;
    } else
           cout << "destroy succeeded" << endl;
}
```

exists

Purpose: Determine whether an object exists.

Given a booted image, determine whether a managed object exists.

```
Boolean exists()
```

Calling Sequence

```
...
Image thing = Image(dn, clsnm);
...
thing.boot(); // Before you call exists(), must boot the image.
if ( !thing.exists() ) {
    cout << "The object does not exist.\n" ;
}
```

This function can return false on an existing object before the first `boot`, since at that point the value of `EXISTS` is `MAYBE`.

The following function call is equivalent to the above function call:

```
if ( !thing.get_prop(“EXISTS”) == ”YES” ) {
```

find_by_nickname

Purpose: Find an image by its nickname.

Returns the value Image() if not found, which in a conditional evaluates to false. All image nicknames are kept in a global registry.

```
static Image find_by_nickname( CDU <name>,
     Platform& <plat> = Platform::def_platform)
```

Calling Sequence

```
...
CDU nicnam = "Server" ;
Image thing = **Image::find_by_nickname( nicnam )** ;
if ( !thing ) {
    cout << "object not found" << endl;
    return 1 ;
}
```

find_by_objname

Purpose: Find an image by its object name.

Returns the value `Image()` if not found, which in a conditional evaluates to false. All image object names are kept in a global registry.

```
static Image find_by_objname( CDU <name>,
     Platform& <plat> = Platform::def_platform)
```

Calling Sequence

```
...
CDU objnam = "Server" ;
Image thing = **Image::find_by_objname( objnam )** ;
if ( !thing ) {
    cout << "object not found" << endl;
    return 1 ;
}
```

To guarantee that the image is created if not found, use the `Image(`*<objname>*`)` constructor.

first_album

Purpose: Return the first album containing the image.

The return value is, in the form of an `AlbumImage` iterator.

```
AlbumImage first_album()
```

The `AlbumImage::next_album` function returns a next album until there are no more.

get

Purpose: Return the value of the attribute,

The attribute is formatted in data language, according to the implicit syntax of the attribute.

```
DU get( CDU <name>, FBits <fb> = 0) const
```

Calling Sequence

```
Image thing ;
 ..
thing.boot() ;                  // thing must be complete & booted.
CDU atnam = "abcXYZ" ;             // Some existing attribute name
CDU atval = thing.get( atnam ) ;
```

The syntax can specify either a list or a scalar. If the attribute is not present, a value of `DU()` is returned.

get_attr_names

Purpose: Get attribute names.

Return an array containing the names the attributes for this image. The image must be valid and booted before calling `get_attr_names()`.

```
Array <CDU> get_attr_names(boolean <all>)
```

If <*all*> is:

- FALSE, returns the names of attributes that currently have values in the image.
- TRUE, returns the names of all attributes that are defined for the object class.

You can examine the EXISTS attribute property to see if the attribute exists in this image.)

Calling Sequence

```
thing.boot(); // thing must be a booted image before this next
call.
Array(DU) attr_names = thing.get_attr_names();
```

For a complete example, see the `sample/album.cc` file.

get_attr_prop

Purpose: Get an attribute property.

Return a property key value of an attribute of the current image.

```
DU get_attr_prop( CDU <attrname>, CDU <key>)
               <attrname> is an attribute name
           <key> is a property key
           Return value is a property key value
```

For instance, the property key TRACKMODE can have any one of the property key values IGNORE, SNAP, or TRACK. If *<key>* does not specify an existing property, a null DataUnit is returned, which tests false in a conditional.

Calling Sequence

```
Image thing ;
 ..
thing.boot() ;                  // thing must be complete & booted.
CDU atnm = "abcXYZ" ;        // Some valid, existing attribute name
CDU propkey = "TRACKMODE" ;      // Or: CDU propkey = duTRACKMODE
DU propval = get_attr_prop( atnm, propkey ) ;
```

Some common property keys and values are listed in Table 3-13.

Table 3-13 Properties Supported by Most Image Attributes

Property Key	Values	Description
TRACKMODE	IGNORE, SNAP, TRACK	This tells the PMI how to track an attribute. Track attributes are maintained by monitoring attribute change events from the MIS. IGNORE attributes are assumed to be uninteresting and never fetched. Attempts to fetch an ignored attribute result in the Invalid exception.
MOD_PENDING	IGNORE, REPLACE, INCLUDE, EXCLUDE, DEFAULT	This tells the PMI how to modify this attribute at the next store or start_store. IGNORE is the initial value, indicating that this attribute is not to be modified. REPLACE is the default set operation, and requests the MIS to do simple assignment using the supplied value. INCLUDE and EXCLUDE request the MIS to perform set inclusion or exclusion on a multi-valued attribute. (On a single-valued attribute, produces the Invalid exception.) DEFAULT requests the MIS to set the attribute to its default value.
IGNORE_ALLOWED REPLACE_ALLOWED INCLUDE_ALLOWED EXCLUDE_ALLOWED DEFAULT_ALLOWED	YES, NO	These read-only properties say whether you can perform the corresponding modification to this attribute. They are meaningful only if the MIS supplies the corresponding information. It's possible for an attribute to claim to be modifiable and yet the MIS refuses to modify it.
MODIFIABLE	YES, NO	This read-only property is true if any of REPLACE, INCLUDE, EXCLUDE, or DEFAULT are allowed.
EXISTS	YES, NO, MAYBE	This read-only property says whether this attribute is known to exist in the managed object represented by the image. Attributes with a TRACKMODE of IGNORE can remain in MAYBE state even though other attributes have been fetched.

get_attr_trackmode

Purpose: Get the attribute TRACKMODE.

Return the TRACKMODE property value for an attribute of the current image.

```
DU get_attr_trackmode( CDU <attrname> )
```

Calling Sequence

```
Image thing ;
 ..
thing.boot() ;                   // thing must be complete & booted
CDU atnm = "abcXYZ" ;        // Some valid, existing attribute name
DU trkmd = thing.get_attr_trackmode( atnm ) ;
```

The following function call is equivalent to the one above:

```
DU trkmd = thing.get_attr_prop(<atnm>, “TRACKMODE”) ;
```

get_attr_numerrors

Purpose: Get the number of attributes with outstanding errors.

Returns the number of attributes that have outstanding error messages resulting from the latest attempt to store.

```
U32 get_attr_numerrors()
```

Calling Sequence

```
Image im ;
U32 n ;
 ...
n = im.get_attr_numerrors() // The im image must be booted.
```

get_attr_last_error

Purpose: Get last error message for this attribute.

Returns the error message associated with the last exception thrown by this attribute, or returned by the MIS.

```
DU get_attr_last_error(CDU <attrname>)
```

Calling Sequence

```
Image im ;
CDU atnm = "abcXYZ" ; // Some valid attribute name
DU lem ;
lem = im.get_attr_last_error( atnm ) ; // im must be booted.
```

get_dbl

Purpose: Get the attribute as a `double`.

Returns the value of the attribute formatted as a `double`. The `Syntax` must be a scalar and be consistent with a `double` representation. If the attribute is not present, a value of 0.0 is returned.

```
double get_dbl( CDU <attrname> )
```

Calling Sequence

```
Image im ;
CDU atnm = "abcXYZ" ; // Some valid attribute name
double attrvalu = im.get_dbl( atnm ) ; // im must be booted.
```

get_gint

Purpose: Get the attribute as a `GenInt`.

Returns the value of the attribute formatted as a `GenInt` (arbitrarily long integer). The `Syntax` must be a scalar and must be consistent with a `GenInt` representation. If the attribute is not present, a value of `GenInt` is returned.

```
GenInt get_gint( CDU <attrname> )
```

Calling Sequence

```
Image im ;
CDU atnm = "abcXYZ" ; // Some valid attribute name
GenInt attrvalu = im.get_gint( atnm ) ; // im must be booted.
```

get_long

Get the attribute as a `long`.

Returns the value of the attribute formatted as a `long`. The `Syntax` must be a scalar and be consistent with a long representation. If the attribute is not present, a value of 0 is returned.

```
long get_long( CDU <attrname> )
```

Calling Sequence

```
Image im ;
CDU atnm = "abcXYZ" ; // Some valid attribute name
long attrvalu = im.get_long( atnm ) ; // im must be booted.
```

get_nickname

Purpose: Get nickname of image.

Returns the nickname of the current image.

```
DU get_nickname()
```

The following function call is equivalent to the one above:

```
DU get_prop(“NICKNAME”) ;
```

Calling Sequence

```
Image im ;
CDU ninm = im.get_nickname() ; // im must be booted.
```

get_objclass

Purpose: Get class of object.

Returns the name of the class of the managed object represented by this image.

```
DU get_objclass()
```

The following function call is equivalent to the one above:

```
DU get_prop("OBJCLASS") ;
```

Calling Sequence

```
Image im ;
DU obcl = im.get_objclass() ; // im must be booted.
```

get_objname

Purpose: Get object name.

Returns the full name of the managed object represented by this image in local distinguished name (LDN) format.

```
DU get_objname()
```

The following is equivalent to the above expression:

```
DU get_prop("OBJNAME") ;
```

Calling Sequence

```
Image im ;
DU obnm = im.get_objname() ; // im must be booted.
```

get_param_syntax

Purpose: Get the syntax for a method parameter.

```
Syntax get_param_syntax( CDU <name>)
```

For more information, refer to `call` and `start` under the description of `Image`, which starts on page 3-59.

Calling Sequence

For the (arbitrarily chosen) object `MDR` and method `getOidName`:

```
Image im ("metaname=\"MDR\"") ;
im.boot() ;                     // im must be complete and booted.
Syntax sntx ;
sntx = im.get_param_syntax( DU ("getOidName") ) ;
```

get_prop

Purpose: Get a property of the current image.

```
DU get_prop( CDU <key>)
    <key> is a property key
    Return value is a property key value
```

For instance, If you specify the property `TRACKMODE`, and there is such a property, then the function returns one of the values `IGNORE`, `SNAP`, or `TRACK`.

If you specify a <*key*> that does not match an existing property, then the function returns a null `DataUnit`, which tests false in a conditional.

Calling Sequence

```
Image thing ;
 ...
thing.boot() ;              // thing must be complete and booted.
CDU propkey = "TRACKMODE" ;    // You can use duTRACKMODE
DU propval = thing.get_prop( propkey ) ;
```

Most images support at least the following properties:

Table 3-14 Properties Supported by Most Images

Properties	**Description**
`OBJNAME`	The full name of the managed object represented by this image. To find an existing object, you must set either this property or the `NICKNAME`.
`OBJCLASS`	The name of the class of the managed object represented by this image. You must set this property in order to create an object.
`NICKNAME`	The nickname of this image.
`TRACKMODE (SNAP, TRACK)`	How the PMI keeps track of the attribute value. `SNAP` images are fetched when the image is booted, and then left alone. `TRACK` images are maintained by monitoring events from the MIS.
`STATE (DOWN, BOOT, UP, SHUTDOWN)`	The current state of the image. Read only. Initially, an image is `DOWN`. When you call `boot` or `start_boot`, the image enters `BOOT` state until the boot is done. The image then remains in the `UP` state until a `shutdown` or a `start_shutdown` is done. It remains in `SHUTDOWN` state until the shutdown is complete, at which time it is `DOWN` again.
`EXISTS (YES, NO, MAYBE)`	Whether the object as a whole is real or imaginary. Before the first `boot`, the state is indeterminate. Real objects cannot be created (unless you enjoy getting the Conflict exception). Imaginary objects can only be created or used as a reference object for a `set_from_ref`. Attempting to get a real attribute value from an imaginary object results in the noshed exception.

get_raw

Purpose: Get the attribute value, encoded form.

Returns the attribute value in an MIS-specific encoded form.

```
Morf get_raw( CDU <attrname>)
```

Calling Sequence

```
Image thing ;
 ...
thing.boot() ;              // thing must be complete and booted.
CDU attrnam = "abcXYZ" ;       // Some valid attribute name
Morf attrval = thing.get_raw( attrnam ) ;
```

get_result_syntax

Purpose: Get the syntax for a method result.

```
Syntax get_result_syntax( CDU <name>)
```

Calling Sequence

```
Image thing ;
 ...
thing.boot() ;              // thing must be complete and booted.
CDU methnam = "abcXyz" ;       // Some valid method name
Syntax sntx = thing.get_result_syntax( methnam ) ;
```

For more information, refer to `call` and `start` under the description of `Image`, which starts on page 3-59.

get_set

Purpose: Gets *imaginary* value from last `set` rather than *real* value.

The `get_set` function is like `get`, but it returns the *imaginary* value from the last `set` rather than the *real* attribute value.

```
DU get_set( CDU <attrname>, FBits <fb> = 0)
```

Calling Sequence

```
Image thing ;
 ..
thing.boot() ;                    // thing must be complete & booted.
CDU atnam = "abcXYZ" ;               // Some existing attribute name
DU atval = thing.get_set( atnam ) ;
```

get_set_dbl

Purpose: Get *imaginary* double value from last `set`.

The `get_set_dbl` function is like `get_dbl`, but it returns the *imaginary* value from the last `set` rather than the *real* attribute value.

```
double get_set_dbl( CDU <attrname>)
```

Calling Sequence

```
Image thing ;
 ..
thing.boot() ;                    // thing must be complete & booted.
CDU atnam = "abcXYZ" ;               // Some existing attribute name
dbl atval = thing.get_set_dbl( atnam ) ;
```

get_set_gint

Purpose: Like `get_gint`, but returns the *imaginary* value.

The `get_set_gint` function is like `get_gint`, but it returns the *imaginary* value from the last `set` rather than the *real* attribute value.

```
GenInt get_set_gint( CDU <name>)
```

Calling Sequence

```
Image thing ;
 ..
thing.boot() ;                  // thing must be complete & booted.
CDU atnam = "abcXYZ" ;             // Some existing attribute name
GenInt atval = thing.get_set_gint( atnam ) ;
```

get_set_long

Purpose: Like `get_long`, but returns the *imaginary* value.

The `get_set_long` function is like `get_long`, but it returns the *imaginary* value from the last `set` rather than the *real* attribute value.

```
long get_set_long( CDU <name>)
```

Calling Sequence

```
Image thing ;
 ..
thing.boot() ;                  // thing must be complete & booted.
CDU atnam = "abcXYZ" ;             // Some existing attribute name
long atval = thing.get_set_long( atnam ) ;
```

get_set_raw

Purpose: Like `get_raw`, but returns the *imaginary* value from the last `set`.

The `get_set_raw` function call is like `get_raw`, but returns the *imaginary* value from the last `set` rather than the *real* attribute value.

```
Morf get_set_raw( CDU <name>)
```

Calling Sequence

```
Image thing ;
 ..
thing.boot() ;                    // thing must be complete & booted.
CDU atnam = "abcXYZ" ;                  // Some existing attribute name
Morf atval = thing.get_set_raw( atnam ) ;
```

get_set_str

Purpose: Like `get_str`, but returns the *imaginary* value from the last `set`.

The `get_set_str` function is like an ordinary `get_str`, but returns the *imaginary* value from the last `set` rather than the *real* attribute value.

```
DU get_set_str( CDU <name>, FBits <fb> = 0)
```

Calling Sequence

```
Image thing ;
 ..
thing.boot() ;                    // thing must be complete & booted.
CDU atnam = "abcXYZ" ;                  // Some existing attribute name
DU atval = thing.get_set_str( atnam ) ;
```

get_str

Purpose: Get attribute as a string.

Returns the value of the attribute formatted as a string without quotes.

```
DU get_str( CDU <attrname>, FBits <fb> = 0)
```

Calling Sequence

```
Image thing ;
 ..
thing.boot() ;                  // thing must be complete & booted.
CDU atnam = "abcXYZ" ;             // Some existing attribute name
DU atval = thing.get_str( atnam ) ;
```

The Syntax must be a scalar and be consistent with a string representation. It is legal to get a numeric value as a string, it is converted for you. In fact, it's legal to get_str anything. If get_str doesn't know anything special to do with the data, it just calls get for you.

The default format bits (0) sometimes produce strings containing newline characters. You might want to suppress this by passing an <*fb*> argument of OMIT_NEWLINES.

By default, the choice specifier is not returned as part of the string for the get_str() function. USE_EXPLICIT_CHOICE should be used as the second argument of this function if you want the choice specifier in the returned value.

If you've registered a Coder for this attribute (or in the absence of that, for the Syntax of this attribute), then that Coder is used to decode the attribute in preference to the standard decoder. (The value need not be a scalar value in this case.) For more information, refer to set_attr_coder under the description of the Platform class, set_coder, under the description of the Syntax class and to the *Solstice Enterprise Manager Application Development Guide*.

get_userdata

Purpose: Get data the application stored under the <*key*> specified.

Returns any data the application might have stored under the <*key*> specified. There are no predefined values.

```
DU get_userdata( CDU <key>)
```

Calling Sequence

```
Image thing ;
 ..
thing.boot() ;                  // thing must be complete & booted.
CDU atnam = "abcXYZ" ;             // Some existing attribute name
DU atval = thing.get_userdata( atnam ) ;
```

If there is no data under that <*key*> for this instance, the return value (a null `DataUnit`) evaluates to false.

get_when_syntax

Purpose: Get the syntax of a given event type.

```
Syntax get_when_syntax( CDU <eventname>)
```

Calling Sequence

```
Image thing ;
 ..
thing.boot() ;                  // thing must be complete & booted.
CDU evnam = "abcXYZ" ;              // Some existing event name
Syntax sntx = thing.get_when_syntax( evnam ) ;
```

For a list of events, refer to the description of `when`, under the description of the `Image` class.

is_in_album

Purpose: Determine whether the image is in the album.

Returns true if the image is contained in the album.

```
Boolean is_in_album( Album <album>)
```

Calling Sequence

```
Image thing ;
Album myalbum ;
 ...      // Before the next call, derive myalbum.
DU atval = thing.is_in_album( myalbum ) ;
```

U32 num_albums

Purpose: Get the number of albums that contain this image.

```
U32 num_albums()
```

Calling Sequence

```
Image thing ;
thing.boot() ;
 ...
U32 na = thing.num_albums() ;
```

revert

Purpose: Cancel any pending `sets`.

Cancels any pending `sets` that have not yet been stored.

```
Result revert()
```

Calling Sequence

```
Image thing ;
thing.boot() ;
 ...
thing.revert() ;
```

send_event

Purpose: Send event notifications to the MIS.

- Send an event to the MIS with a custom timestamp.

```
Result send_event (DU <event_name>,
           DU <event_info>=DU(),
           struct tm *<tt>)
```

- Send an event to the MIS with the timestamp expressed in ASN.1.

```
Result send_event (DU <event_name>,
           Asn1Value &<event_info>,
           Asn1Value &<time>=Asn1Value())
```

- Send an event to the MIS with the default timestamp of *now*.

```
Result send_event (DU <event_name>,
               struct tm *<tt>)
```

See the sample programs `sample/event_send1.cc`, `event_send2.c`, and `event_send3.cc` for details.

The image used in the `send_event()` call is the representation of the managed object that is generating the event notification (such as an alarm). This notification is sent to the MIS, from where it is forwarded to applications that are interested in events of this type.

The image instance (of which `send_event()` is a member) is the generator of the event. In using `send_event()`, a PMI application is acting in an agent role, since only agents generate events.

Requirements: Before calling `send_event()`, keep in mind that:

- The image must be of a valid object class and have a valid name;
- The syntax of the argument <*event_info*> is based on the event and is parsed accordingly.

When the value of <*tt*> is zero, the event is sent with the timestamp of "Now". Timestamp fields follow standard Unix conventions:

```
Timestamp sent =
(tt->tm_year + 1900, tt->tm_mon + 1, tt->mday, tt->tm_hour,
 tt->tm_min, tt->tm_sec);
```

set

Purpose: Set attributes.

The `set` function encodes the textual data you pass and modifies the value of the attribute using the encoded value.

```
Result set( CDU <name>,

            CDU <val>,

            CDU <op> = duREPLACE,

            FBIts <fb> = 0)
```

Actually, it changes the *imaginary* value of the attribute. The *real* value is not changed until the next `store`. The data language is interpreted according to the syntax already implicit in the attribute. If the data cannot be so interpreted, the Invalid exception is thrown. The syntax can specify either a list or a scalar. A series of `set` operations can be undone before the `store` by calling `revert`. Refer to the `MOD_PENDING` attribute property in Table 3-13 on page 3-73 for a description of legal operations.

set_attr_prop

Purpose: Set a property of an attribute.

Sets a property of an attribute of the current image.

```
Result& set_attr_prop( CDU <name>,
    CDU <key>,
     CDU <value>)
```

Requirements:

- <*name*> specifies an existing attribute in the object class
- <*key*> specifies a supported property
- <*value*> specifies a legal value.

If `set_attr_prop` cannot do what you ask, it throws the Invalid exception. Refer to `get_attr_prop`, under the description of `Image`, which starts on page 3-59, for some typical properties.

set_dbl

Purpose: Modify an attribute using a double.

Encodes the double you pass and modifies the value portion of the attribute using the encoded value.

Actually, it changes the *imaginary* value of the attribute. The *real* value is not changed until the next `store`. The syntax of the attribute must be a scalar and permit a double representation, or the Invalid exception is thrown.

```
Result set_dbl( CDU <name>,
    double <val>,
    CDU <op> = duREPLACE)
```

Refer to the `MOD_PENDING` attribute property in Table 3-13 on page 3-73 for a description of legal operations.

set_from_ref

Purpose: Copy attribute values from an object to its current image.

Copies the attribute values from the reference object into the current image.

```
Result set_from_ref( Image& <refobj>)
```

The state of the reference object must be `UP`, but the reference object need not exist. If the reference object exists, then its *real* attributes are copied. Otherwise its *imaginary* attributes are copied. Attributes that receive existing values are automatically given a `MOD_PENDING` property of `REPLACE`. Attributes that receive nonexistent values are given a `MOD_PENDING` property of `IGNORE`.

set_gint

Purpose: Modify an attribute using a `GenInt`.

Encodes the arbitrarily long integer you pass and modifies the value portion of the attribute using the encoded value. Actually, it changes the *imaginary* value of the attribute. The *real* value is not changed until the next `store`.

```
Result set_gint( CDU <name>,
    GenInt& <val>,
    CDU <op> = duREPLACE)
```

The syntax of the attribute must be a scalar and permit a `GenInt` representation, or the Invalid exception is thrown. Refer to the `MOD_PENDING` attribute property in Table 3-13 on page 3-73 for a description of legal operations.

set_long

Purpose: Modify an attribute using a `long`.

Encodes the `long` you pass and modifies the value portion of the attribute using the encoded value. Actually, it changes the *imaginary* value of the attribute. The *real* value is not changed until the next `store`.

```
Result set_long( CDU <name>,
    long <val>,
    CDU <op> = duREPLACE)
```

The syntax of the attribute must be a scalar and permit a long representation, or the Invalid exception is thrown.

Refer to the `MOD_PENDING` attribute property in Table 3-13 on page 3-73 for a description of legal operations.

set_nickname

Purpose: Set the nickname for an image.

```
Result set_nickname( CDU <nickname>)
```

The following is equivalent to the above:

```
Result set_prop("NICKNAME",<nickname>) ;
```

set_objclass

Purpose: Set the object class for an image.

```
Result set_objclass( CDU <name>)
```

The following is equivalent to the above:

```
Result et_prop("OBJCLASS",<name>) ;
```

set_prop

Purpose: Set a property of the current image.

Requirements:

- <*key*> specifies a supported property
- <*value*> specifies a legal value

```
Result set_prop( CDU <key>,

                 CDU <value>)
```

If `set_prop` cannot do what you ask, it throws the Invalid exception. Refer to `get_prop` under the description of the `Album` class, which starts on page 3-17, for some typical properties.

set_raw

Purpose: Load data into the *imaginary* value of an attribute.

The `set_raw` function loads a MIS-specific, encoded data value into the *imaginary* value of the attribute. If you later do a `store`, that updates the *real* value of the attribute.

```
Result set_raw( CDU <attrname>,
    Morf& <val>,
    CDU <op> = duREPLACE)
```

Refer to the `MOD_PENDING` attribute property in Table 3-13 on page 3-73 for a description of legal operations.

set_str

Purpose: Encode a string and modify the attribute.

The `set_str` function encodes the string you pass and modifies the value portion of the attribute using the encoded value.

Actually, it changes the *imaginary* value of the attribute. The *real* value is not changed until the next `store`. The syntax of the attribute must be a scalar and permit a string representation, or the Invalid exception is thrown. For choice type attributes, a choice specifier should be used.

```
Result set_str( CDU <attrname>,
    CDU <val>,
    CDU <op> = duREPLACE)
```

Calling Sequence

```
Image im ;
im.boot() ;
char dn[300]            = "logId=\"AlarmLog\"";
char class_name[300]    = "log";
char attribute_name[300]= "maxLogSize";
char set_val[300]       = "666666";
 ...
if(!im.set_str(attribute_name, set_val)) {
...
```

See the `/opt/SUNWconn/em/src/sample/set.cc` file for a full example.

The difference between `set_str` and `set` is that the data language used by `set` requires quotes as part of the string, while `set_str` assumes them if necessary. (They're not always necessary; you can also pass numeric values as strings, and they are converted for you.) Refer to the `MOD_PENDING` attribute property in Table 3-13 on page 3-73 for a description of legal operations.

If you've registered a `Coder` for this attribute (or in the absence of that, for the `Syntax` of this attribute), then that `Coder` is used to encode the attribute in preference to the standard encoder. (The value need not be a scalar value in this case.) For more information, refer to `set_attr_coder` under the description of the `Platform` class, which starts on page 3-109, `set_coder`, under the description of the `Syntax` class and to the *Solstice Enterprise Manager Application Development Guide*.

set_userdata

Purpose: Store data supplied by the application.

The `set_userdata` function stores arbitrary data supplied by the application, under the <*key*> specified.

```
Result set_userdata( CDU <key>, CDU <value>)
```

There are no predefined values. If there was already data under that <*key*> for this instance, it is replaced without comment. Essentially, this is an associative array belonging to the album that the application can use any way it pleases.

shutdown

Purpose: Deactivate an image.

The `shutdown` function deactivates an image and invalidates all locally cached attribute values.

```
Result shutdown( Timeout <to> = DEFUALT_TIMEOUT)
```

A tracking image stops tracking. When complete, the image's `STATE` is set to `DOWN`.

start

Purpose: Provide an asynchronous version of `call`.

The `start` function is the asynchronous version of `call`. It sends an unconfirmed action request message when callback is NO_CALLBACK (the default).

```
Waiter start( CDU <name>, CDU <param>, CCB <cb> = NO_CALLBACK)
```

start_boot

Purpose: Provide an asynchronous version of `boot`.

```
Waiter start_boot( CCB <cb> = NO_CALLBACK)
```

start_create

Purpose: Provide an asynchronous version of `create`.

```
virtual Result start_create( Image& <refobj> = Image(),

CCB <cb> = NO_CALLBACK)
```

start_create_within

Purpose: Provide an asynchronous version of `create_within`.

```
Waiter start_create_within( CDU <objname>,
    Image& <refobj> = Image(),
    CCB <cb> = NO_CALLBACK)
```

start_destroy

Purpose: Provide an asynchronous version of `destroy`.

```
Result start_destroy( CCB <cb> = NO_CALLBACK)
```

start_raw

Purpose: Provide an asynchronous version of `call_raw`.

```
Waiter start_raw( CDU <name>,
    Morf <param>,
    CCB <cb> = NO_CALLBACK)
```

start_shutdown

Purpose: Provide an asynchronous version of `shutdown`.

```
Waiter start_shutdown( CCB <cb> = NO_CALLBACK);
```

start_store

Purpose: Provide an asynchronous version of `store`.

```
Waiter start_store( CCB &<cb> = NO_CALLBACK)
```

All `start_store set` requests are sent in unconfirmed mode when `CCB` is equal to `NO_CALLBACK` (the default).

store

Purpose: Update actual attributes using imaginary attributes.

The `store` function updates the actual object's attributes using any imaginary attributes that have been created by `set` and any others.

```
Result store( Timeout <to> = DEFAULT_TIMEOUT)
```

See also `start_store()`.

when

Purpose: Establish a callback routine.

The `when` function establishes a callback routine to handle an image-specific asynchronous event.

```
Result when( CDU <eventname>,
    CCB <cb> = NO_CALLBACK)
```

The `Platform` object receives all events at which time all callbacks registered for by the `Platform` objects are executed. Next, all of the callbacks registered for by image objects are executed, then all of the callbacks registered for by album objects are executed.

For example, you might want to know if an attribute of the image changed. You might say:

```
when(“ATTR_CHANGED”, Callback(attr_change_cb, 0)) ;
```

`Image` Events include:

Table 3-15 Image-specific Asynchronous Events

Events	**Description**
`OBJECT_CREATED`	The object represented by this image was successfully created (not necessarily by us!).
`OBJECT_DESTROYED`	The object represented by this image was successfully destroyed (not necessarily by us!).
`ATTR_CHANGED`	An attribute of the object represented by this image has changed in value.
`RAW_EVENT`	Any object-related event can be examined as a raw event before ordinary event processing by the PMI.

3.7.10 `Morf` *Class*

Inheritance: `public Error`

`#include <pmi/hi.hh>`

Data Members: No public data members declared in this class

A `Morf` is a reference-counting wrapper around an abstract base class. Each framework derives a new class from the base class and provides the implementation for manipulating that type of data in the context of the framework. Each instance of the derived class contains an opaque, encoded value along with the information necessary for the PMI to be able to decode it.

As noted in their individual descriptions, some of this class's methods require a list argument, some a scalar, and some accept either.

Table 3-16 Morf Method Types

Method Name	**Data Domain**	**Method Type**
void* = == !=		Operator Overloading

`get_memname`
`get_platform`
`get_syntax`
`has_value`
`ref`
	Scalar or list	Get information about an existing Morf

`is_choice` `is_list`	Scalar or list	Distinguish syntax types

`extract`
`get_member_names`
`num_elements`
`split_array`
`split_queue`
	List only	Pull apart list Morfs

`set`	Scalar or list	Set the data value of an existing Morf

`set_dbl`
`set_gint`
`set_long`
`set_str`
set_value
	Scalar only
	
	Scalar or list
		Set the data value of an existing Morf

Table 3-16 Morf Method Types

Method Name	**Data Domain**	**Method Type**
`get`	Scalar or list	Get the data value from an existing Morf

`get_dbl`
`get_gint`
`get_long`
`get_str`
get_value
	Scalar only
	
	Scalar or list
		Get the data value from an existing Morf

`get_memname` `set_memname`	Choice only	

For more information on how to use the `Morf` class, refer to the discussion of the `em_cmipconfig` example in the "Examples" chapter of the *Solstice Enterprise Manager Application Development Guide* (this book is provided on AnswerBook only).

3.7.10.1 Constructors

```
Morf()
```

The default constructor creates a `Morf` instance that refers to no actual `Morf`. The value tests false until you assign it a real `Morf` value.

```
Morf(Syntax& <syn>)
```

The preceding constructor constructs a `Morf` instance for a particular kind of syntax. Because a `Morf` is really a wrapper for a set of related classes, this function actually works like a virtual constructor.

```
Morf(const Morf& <other>)
```

The preceding constructor is an ordinary copy constructor. After the copy, both copies still refer to the same `Morf` object. The reference count on the `Morf` object is incremented.

```
Morf(Syntax& <syn>, CDU <data>)
```

The preceding constructor constructs a `Morf` instance for a particular kind of syntax, which implies a particular kind of MIS. Because a `Morf` is really a wrapper for a set of related classes, this function actually works something like a virtual constructor. The textual data supplied as the second argument is parsed according to the syntax supplied, so you can create either scalar or list `Morf`s with this function.

```
Morf(Syntax& <syn>, Array<Morf>& <ma>)
```

The preceding constructor constructs a `Morf` instance for a particular kind of syntax. Because a `Morf` is really a wrapper for a set of related classes, this function actually works rather like a virtual constructor. Because an array of `Morf`s is supplied as the second argument, only list Morfs can be created with this function.

```
Morf(Syntax& <syn>, Queue<MorfElem>& <mq>)
```

The preceding constructor constructs a `Morf` instance for a particular kind of syntax. Because a `Morf` is really a wrapper for a set of related classes, this function actually works a bit like a virtual constructor. Because a queue of `MorfElem`s is supplied as the second argument, only list `Morf`s that can be created with this function.

```
Morf( CDU <attrname>, Platform& <plat> = Platform::def_platform)
```

The preceding constructor constructs a `Morf` instance for a particular kind of attribute, which implies a particular syntax. One could conceivably accomplish this by creating an image for an object of the type containing the attribute in question, and then extracting the `Morf` corresponding to that attribute, but this way is easier.

```
Morf( Ptr <ptrdata>, Boolean <reuse> = FALSE)
```

The preceding constructor constructs a `Morf` instance from a `void*` pointer created by the `ref` method. It is primarily for internal PMI use within callbacks, when the callback can occur after the original `Morf` has gone out of scope, and would ordinarily have been deleted. Each call to `ref` increments a reference count, and each construction of a `Morf` using this constructor eventually causes the reference count to be decremented again when the `Morf` is destructed at the end of the callback. If multiple callbacks are to use the same pointer, then pass a `reuse` parameter of `TRUE` on all but the last callback (or just call `ref` again within the callback) to keep the reference alive till the next callback.

3.7.10.2 Operator Overloading for Morf

```
Morf& operator = (const Morf& <other>)
```

The assignment operator works just like the copy constructor.

```
operator void*()
```

The preceding cast operator is for use in conditionals. It returns true if this `Morf` refers to an actual `morf` object. Do not attempt to use the returned value as a pointer to anything, since it points to private data.

```
int operator !()
```

The preceding function is provided so that you can say “`if (!morf)`…”

```
int operator == (const Morf& <other>)
```

The preceding comparison operator returns true if the two compared `Morf`s are equivalent in value. The definition of “equivalent” depends on the system type of the MIS. Some values might be impossible to compare. In this case the MIS-specific code is allowed either to return false or throw an exception, depending on how miffed it feels at the time.

```
int operator != ( const Morf& <other>)
```

The preceding comparison operator returns true if the two compared `Morf`s are *not* equivalent in value.

3.7.10.3 Member Functions of Morf

This section describes the member functions of the `Morf` class.

extract

```
Morf extract( DU <navigation>)
```

The preceding function call can navigate down a tree of `Morf`s and pull out a particular `Morf`. The <*navigation*> parameter is a string containing a dot-separated list of field names or position numbers. By convention, position 0 returns the number of elements in the list at that level of the tree.

Valid only for a `Morf` that describes a list of values (or a choice). If called on a scalar `Morf` it throws the Invalid exception. When called on a choice value with a null <*navigation*> string, it returns a `Morf` of the current inner type rather than the outer choice type.

get

```
DU get() const
```

The preceding function call returns the value of the `Morf` formatted in data language according to the implicit `Syntax` of the `Morf`. The `Syntax` can describe either a list or a scalar. Various format bits can be OR'ed together to influence the form of output. If the `Morf` has no value, `DU()` is returned.

get_dbl

```
double get_dbl()
```

The preceding function call returns the value of the `Morf` formatted as a double. The `Syntax` must describe a scalar and be consistent with a double representation. Otherwise the Invalid exception is thrown. If the `Morf` has no value, 0.0 is returned.

get_gint

```
GenInt get_gint()
```

The preceding function call the value of the `Morf` formatted as a `GenInt` (arbitrarily long integer). The `Syntax` must describe a scalar and be consistent with a `GenInt` representation. Otherwise the Invalid exception is thrown. If the `Morf` has no value, `GenInt()` is returned.

get_long

```
long get_long()
```

The preceding function call returns the value of the `Morf` formatted as a long. The `Syntax` must describe a scalar and be consistent with a long representation. Otherwise the Invalid exception is thrown. If the `Morf` has no value, 0 is returned.

get_member_names

```
Array<DU> get_member_names()
```

The preceding function call returns the member names (field names) for a `list` value. For a `choice` value, it returns the member names of an inner list, presuming that the current value of the `choice` is a `list` value. To get the `choice` names themselves, you must use `get_syntax` and `get_member_names` instead.

get_memname

```
DU get_memname()
```

The preceding function call returns the name of the member (field) currently held by the `Morf`. Valid only for members of lists and choices. (The `Morf` itself need not be a list or choice.)

get_platform

```
Platform get_platform()
```

The preceding function call returns the `Platform` of the `Syntax` that is implicitly bound into the `Morf`. Valid either for a `Morf` that describes a scalar value or a list of values.

get_str

```
DU get_str( FBits <fb> = 0)
```

The preceding function call returns the value of the `Morf` formatted as a string (without quotes). The `Syntax` must describe a scalar and be consistent with a string representation. It is legal to get a numeric value as a string; it is automatically converted for you. In fact, any value is legal. If the type is unrecognized, `get_str` just calls `get` for you. If the `Morf` has no value, `DU()` is returned.

The default format bits (0) sometimes produce strings containing newline characters. You might want to suppress this by passing an <*fb*> argument of `OMIT_NEWLINES`.

If you've registered a `Coder` for this `Morf`'s attribute (or in the absence of that, for the `Syntax` of this `Morf`), then that `Coder` is used to decode the value in preference to the standard decoder. (The value need not be a scalar value in this case.) For more information, refer to `set_attr_coder` under the description of the `Platform` class, which starts on page 3-109, and `set_coder`, under the description of the `Syntax` class.

get_syntax

```
Syntax get_syntax()
```

The preceding function call returns the `Syntax` that is implicitly bound into the `Morf`. Valid either for a `Morf` that describes a scalar value or a list of values.

get_value

```
Asn1Value get_value()
```

The preceding function call returns the encoded value stored in the `Morf`, if any.

has_value

```
void* has_value()
```

The preceding function call returns a pointer to the internal, MIS-specific value, if any. Otherwise, returns 0. Note that this differs from operator `void*()`, which can return true even when the `Morf` contains only a `Syntax` with no value.

is_choice

```
Boolean is_choice()
```

The preceding function call returns true if the `Syntax` of the `Morf` describes a choice value. A choice value can have any one of a number of types of value. (The `get_memname` function tells you which kind of value the current `Morf` is holding.) Valid either for a `Morf` that describes a scalar value or a list of values.

is_list

```
Boolean is_list()
```

The preceding function call returns true if the `Syntax` of the `Morf` describes a compound data value, and false if it describes a scalar value. Valid either for a `Morf` that describes a scalar value or a list of values. Note that a list with only one element, or zero elements, is still a list and not a scalar.

num_elements

```
U32 num_elements()
```

The preceding function call returns the number of elements in the `Morf`'s list. Valid only for a `Morf` that describes a list of values.

ref

```
Ptr ref()
```

The preceding function call returns a reference-counted `void*` pointer to this `Morf`, from which you *must*, at some future time, reconstruct the `Morf` using the `Morf(Ptr, Boolean)` constructor. Refer to that constructor description earlier in this section for further information.

set

```
Morf set( CDU <data>, Fbits <fb> = 0)
```

The preceding function encodes the textual data you pass and replaces the value portion of the `Morf` with the encoded value. The data language is interpreted according to the `Syntax` already implicit in the `Morf`. If the data cannot be so interpreted, the Invalid exception is thrown. The `Syntax` can describe either a list or scalar.

set_dbl

```
Morf set_dbl( double <data>)
```

The preceding function call encodes the double you pass and replaces the value portion of the `Morf` with the encoded value. The `Syntax` of the `Morf` must be a scalar and permit a double representation, or the Invalid exception is thrown.

set_gint

```
Morf set_gint(GenInt& <data>)
```

The preceding function call encodes the `GenInt` you pass and replaces the value portion of the `Morf` with the encoded value. (A `GenInt` is an arbitrarily long integer.) The `Syntax` of the `Morf` must be a scalar and permit a `GenInt` representation, or the Invalid exception is thrown.

set_long

```
Morf set_long(long <data>)
```

The preceding function call encodes the long you pass and replaces the value portion of the `Morf` with the encoded value. The `Syntax` of the `Morf` must be a scalar and permit a long representation, or the Invalid exception is thrown.

set_memname

```
Result set_memname()
```

The preceding function call sets the name of the member (field) currently held by the Morf. Valid only for members of a choice type. The old value of the Morf is discarded.

set_str

```
Morf set_str( CDU <data>, FBits <fb> = 0)
```

The preceding function call encodes the string you pass and replaces the value portion of the Morf with the encoded value. The Syntax of the Morf must be a scalar and permit a string representation, or the Invalid exception is thrown.

The difference between set_str and set is that the data language used by set requires quotes as part of the string, while set_str assumes them if necessary. (They're not always necessary; you might also pass numeric values as strings, and they are converted for you.)

If you've registered a Coder for this Morf's attribute (or in the absence of that, for the Syntax of this Morf), then that Coder is used to encode the value in preference to the standard encoder. (The value need not be a scalar value in this case.) For more information, refer to set_attr_coder under the description of the Platform class, which starts on page 3-109, and set_coder, under the description of the Syntax class.

set_value

```
void set_value(Asn1Value <data>)
```

The preceding function call sets the encoded value into the Morf. Checking is not performed on the value.

split_array

```
Array <Morf> split_array()
```

The preceding function call returns the elements of a list `Morf` in `Array` form, such that they can be indexed numerically.

Valid only for a `Morf` that describes a list of values. If called on a scalar `Morf`, it throws the Invalid exception.

split_queue

```
Queue <MorfElem> split_queue()
```

The preceding function call returns the elements of a list `Morf` in `Queue` form, such that they can be processed with ordinary `Queue` commands. Valid only for a `Morf` that describes a list of values. If called on a scalar `Morf` it throws the Invalid exception.

3.7.11 `PasswordTty` *Class*

Inheritance: `none`

`#include <pmi/password_tty.hh>`

Data Members: No public data members declared in this class

This class implements the TTY based password query mechanism. You can derive from this class to implement different password query mechanism (for example, dialog box based for GUI applications). All non-GUI EM applications automatically get the TTY based password query mechanism. The `Platform::connect` method determines if user's password is required and accordingly calls a method on this class to get the password. If you don't wish to alter the default TTY based password query mechanism, you don't need to use this class.

At any given time, there can be at most one instance of this class or one of its derived classes. If no instance exists, then the Platform::connect method temporarily creates one and uses that to query the user's password. If you derive from this class and create an instance of it before you call

`Platform::connect`, then that instance will be used to query the password. This is achieved using the virtual methods in C++. You need to implement the `PasswordTty::password_function` virtual method in your class to replace the default TTY based password query mechanism.

For an example program, please refer to files in $EM_HOME/src/access_passwd which demonstrate how to implement dialog box based password query mechanism in Motif based GUI applications.

3.7.11.1 Constructors

```
PasswordTty ()
```

Creates an instance of the PasswordTty class. This will result in an assertion failure if an instance of this or its derived classes already exists. At any given time, there can be at most one instance of this class or one of its derived classes.

3.7.11.2 Operator Overloading

No public operators are defined for this class.

3.7.11.3 Member functions

split_queue

```
virtual int password_function (char *user, char *password) const
```

This function queries the user for login name and password. The login name defaults to the user argument. This method returns a non-zero value if password query succeeds. If you derive from the PasswordTty class, you should implement this method to provide your own password query mechanism.

3.7.12 `Platform` *Class*

Inheritance: `public Error`

```
#include <pmi/hi.hh>
```

Data Members: No public data members declared in this class

An instance of the `Platform` class represents a potential or actual connection to a particular MIS, along with all the implied semantics of the framework implemented by the MIS. The `Platform` is a reference-counting wrapper around an inner abstract base class; each framework derives a new class from the base class to implement framework-specific semantics. (The base class does provide a generic attribute-like mechanism that specific frameworks can use for specifying things like access tickets or default time-outs.)

For a discussion on how to use the `Platform` class, refer to the "`em_cmipconfig` Example" section of the "Examples" chapter of the *Solstice Enterprise Manager Application Development Guide*.

Table 3-17 Platform Method Types

Method Name	**Method Type**
`default_platform` `set_default_platform`	MIS default

`get_prop`
`set_prop`
replace_discriminator
replace_discriminator_classes
	Property control

`connect`
`disconnect`
`start_connect`
`start_disconnect`
get_connection_fd
	MIS connection

`find_album_by_nickname`
`find_image_by_nickname`
`find_image_by_objname`
`get_fullname`
`get_shortname`
	Name translation

`get_when_syntax` `when`	Function callback

Table 3-17 Platform Method Types

Method Name	**Method Type**
`get_plat_id` `get_raw_sap`	Utility routines
`get_attr_coder` `set_attr_coder`	String encoding / decoding hooks
get_authorized_features get_authorized_applications	Access control

3.7.12.1 Constructors

```
Platform()
```

The default constructor creates a `Platform` instance that refers to no actual MIS. The value tests false until you assign it a real `Platform` value.

```
Platform( const Platform& <plat>)
```

The preceding is an ordinary copy constructor. After the copy, both copies still refer to the same MIS object. The reference count on the MIS object is incremented.

```
Platform( CDU <plattype>, CDU <nickname> = DU() )
```

The preceding constructor constructs a `Platform` instance for a particular kind of MIS. Because a `Platform` is really a wrapper for a set of related classes, this function actually works a bit like a virtual constructor.

3.7.12.2 *Operator Overloading for Platform*

```
Platform& operator = (const Platform& <plat>)
```

The assignment operator, above, works just like the copy constructor.

```
operator void*()
```

The preceding cast operator is for use in conditionals. It returns true if this `Platform` refers to an actual MIS object (regardless of whether that MIS is connected yet). Do not attempt to use the returned value as a pointer to anything, since it points to private data.

```
int operator !()
```

The preceding operator definition is provided so that you can say "`if (!platform)`..."

3.7.12.3 *Member Functions of Platform*

This section describes the member functions of the `Platform` class.

connect

```
Result connect(CDU <location>,
               CDU <application_name>)
```

The preceding function call attempts to connect to the MIS. The value of <*location*> is implementation dependent, but might be something as simple as a host name.

This method determines if the user is subject to password authentication. If he/she is then this method queries the login name and password. By default, a tty-based password query mechanism is used. An application can redefine the password query mechanism by deriving from the `PasswordTty` class and providing its own definition of the `PasswordTty::password_function` virtual method.

The argument <*application_name*> is used by the Access Control Module to determine if the user has permission to use the application. If the user doesn't have permission to use the application this method fails to connect to the MIS and returns NOT_OK. The `Platform::get_error_string` function can be used to determine the reason for the failure.

default_platform

```
static Platform default_platform()
```

The preceding function call returns the default MIS. The `Image` and `Album` constructors use this value if you invoke them without supplying an argument to specify an MIS. To set the default MIS, use `set_default_platform`.

disconnect

```
Result disconnect( double <to> = DEFAULT_TIMEOUT)
```

The preceding function call disconnects the MIS in a clean fashion. The destructor calls this function for you if you don't call it first.

find_album_by_nickname

```
Album find_album_by_nickname( CDU <name>)
```

The preceding function call locates the album that has registered itself with the specified <*name*>. The null value `Album()` is returned if no such album is found.

find_image_by_nickname

```
Image find_image_by_nickname( CDU <name>)
```

The preceding function call locates the image that has registered itself with the specified <*name*>. The null value `Image()` is returned if no such image is found.

find_image_by_objname

```
Image find_image_by_objname( CDU <name>)
```

The preceding function call locates the image that has registered itself with the specified <*name*>. The null value `Image()` is returned if no such image is found. Note that the `Image` constructor can also do lookups for you, should you want the image to be created when not found.

get_attr_coder

```
Coder* get_attr_coder( CDU <attrname>)
```

The preceding method returns the encoder/decoder for a given attribute.

get_authorized_applications

```
Result get_authorized_applications (
   AuthApps &apps,
   const char *user = 0
);
```

This method is used get the list of authorized applications for the given *user*. In most cases, you should use the default value of the *user* argument, which will default to the currently logged in user. The user logged in to MIS may be different than the user running the application since the user name can be changed during the password query.

This method should be called after `Platform::connect` has been successful. This method returns OK if there is no unexpected PMI error and if the user is authorized to use at least one application. You can query if an application is authorized or not by using the `AuthApps::is_authorized()` method on the *authApps* argument.

Please notice, this method uses the `AuthApps` class, whereas, the method `Platform::get_authorized_features` uses the `AuthFeatures` class. If all applications are authorized, any argument to the `AuthApps::is_authorized()` will return OK even if the application has not yet been registered in the MIS.

The following is an example of this function's use:

```
include <pmi/auth_apps.hh>
// After Platform::connect has been successful ...

AuthApps authApps;
if (platform.get_authorized_applications(authApps) == OK) {
        if (!authApps.is_authorized(“em_discover”)) {
            // em_discover is not authorized
        }
        // Check other apps ...
}
else {
        // handle unexpected PMI error
}
```

get_authorized_features

```
Result get_authorized_features (
   AuthFeatures &features,
   const char *user = 0,
   const char *appname = 0
);
```

This method is used get the list of authorized features for the given *user* and the *application*. In most cases, you should use the default value of the *user* and the *application* argument, which will default to the currently logged in user and the current application being run. The user logged in to MIS may be different than the user running the application since the user name can be changed during the password query.

This method should be called after `Platform::connect` has been successful. This method returns OK if there is no unexpected PMI error and if the user is authorized to use at least one feature for the given application. You can query if a feature is authorized or not by using the `AuthFeatures::is_authorized()` method on the *features* argument.

Please notice, this method uses the `AuthFeatures` class, whereas, the method `Platform::get_authorized_applications` uses the `AuthApps` class. If all features are authorized, any argument to the `AuthFeatures::is_authorized()` will return OK even if the feature has not yet been registered in the MIS.

The following is an example of this function's use:

```
#include <pmi/auth_features.hh>
// After Platform::connect has been successful ...

AuthFeatures features;
if (platform.get_authorized_features(features) == OK) {
       // For the sake of this example, it is assumed,
       // there are two features, “Create” and “Delete”, in
       // this application. It is also assumed that by
default these
       // features are enabled. The following code
       // checks if a feature is not authorized and disables
the feature.

       if (!features.is_authorized(“Create”)) {
           // Disable “Create” feature
       }
       if (!features.is_authorized(“Delete”)) {
           // Disable “Delete” feature
       }
}
else {
       // handle unexpected PMI error
}
```

get_connection

```
int get_connection_fd()
```

The preceding method returns the file descriptor corresponding to the connection to an MIS. It is useful in a GUI program for waiting in a `select()` (3C) for platform events.

get_fullname

```
DU get_fullname( CDU <shortname>)
```

The preceding function call translates a short attribute name to its fully qualified name. Refer also to the `Platform::when` method description for more information.

get_plat_id

```
PlatformId get_plat_id()
```

The preceding function call returns the MIS ID number, which is not very useful outside of the PMI.

get_prop

```
DU get_prop( CDU <key>)
```

The preceding function call returns a property of the current MIS. If <*key*> does not specify an existing property, returns a null `DataUnit`, which tests false in a conditional. Most MISs support, at a minimum, the properties in Table 3-18.

Table 3-18 MIS Properties Supported

Properties	**Description**
PLATFORM_TYPE	As specified to the "virtual" constructor. (Read Only)
PLATFORM_OBJNAME	Absolute object name of the MIS itself. (Read Only)
APPLICATION_OBJNAME	Object instance name of the object (within the MIS) that represents the current application's connection and/or process. (Read Only)
PLATFORM_NICKNAME	The nickname for the MIS that you specified to the "virtual" constructor.
APPLICATION_TYPE	The kind of application specified to connect. (Read Only)
LOCATION	The location of the MIS specified to connect. (Read Only)
STATE	The state of the connection (Read Only): DOWN - Unconnected BOOT - Connecting UP - Connected SHUTDOWN - Disconnecting
DEFAULT_TIMEOUT	The default time-out for this MIS, in seconds, with fractions allowed.

If you want to construct an event sieve by hand, you need to know the application instance name to tell the sieve where to forward events. The PMI constructs most sieves for you automatically, so you generally need not be concerned with this.

See the description of the replace_discriminator() and replace_discriminator_classes() methods, below. These methods allow you, in some programs, to do without a call to get_prop().

The PMI makes no use of the nickname property; it is there merely as a convenience.

The synchronous version of any function that has both a synchronous and an asynchronous version uses the default time-out. If you set the default time-out to 0, only the asynchronous version works (unless of course you specify an explicit time-out on the synchronous call itself).

get_raw_sap

```
void* get_raw_sap()
```

The preceding function call returns the message SAP of the internal connection to the MIS. This is useful for programs that need to use both the low and high levels of the PMI, when both need to share the same connection.

get_shortname

```
DU get_shortname( CDU <fullname>)
```

The preceding function call translates a fully qualified attribute name to its corresponding short name. Typically this involves stripping a document name from it.

get_when_syntax

```
Syntax get_when_syntax( CDU <eventname>)
```

The preceding function call returns the syntax of the info that is passed to the callback function. Primarily for internal use.

```
Result replace_discriminator_classes (Array (DU)
<object_classes>,
             Array (DU) <event_types> = DEF_ARRAY_DU)
```

replace_discriminator

```
Result replace_discriminator (DU <discriminatorConstruct>)
```

The first method, above, takes an array of classes, such as `topo` objects, and an array of events, such as `objectCreate`. The second method takes a discriminator construct, for example, `(and:{item:equality...})`. The second method is more generalized and more difficult to use than the first. These two methods work on an application instance object. In certain cases, their use can allow you to avoid a call to `get_prop()`. Their use reduces network traffic and unnecessary processing time.

set_attr_coder

```
void set_attr_coder( CDU <attrname>, Coder* <coder>)
```

The preceding function call sets the encoder/decoder for a given attribute. The `Coder` is used by the `get_str` and `set_str` functions in both `Morf`s and `Image`s. See also `set_coder`, under the description of the `Syntax` class.

set_default_platform

```
static void set_default_platform( Platform& <plat>)
```

The preceding function call sets the default MIS. The `Image` and `Album` constructors use this value when you omit the `platform` argument. You don't usually need to call this function, since the first time you connect to a MIS it is invoked for you, and most applications talk to only one MIS.

set_prop

```
Result set_prop( CDU <key>, CDU <value>)
```

The preceding function call sets a property of the current MIS, provided that the <*key*> specifies a supported property and the <*value*> specifies a legal value. If `set_prop` cannot do what you ask, it throws the Invalid exception. Refer to Table 3-18 on page 3-118 for some typical properties.

start_connect

```
Waiter start_connect( CDU <platform>,
                      CDU <application_name>,
                      CCB <cb> = NO_CALLBACK)
```

The preceding function call is the asynchronous version of `connect`.

start_disconnect

```
Waiter start_disconnect( Callback& <cb> = NO_CALLBACK)
```

The preceding function call is the asynchronous version of `disconnect`.

when

```
Result when( CDU <eventname>, CCB <cb> = NO_CALLBACK)
```

The `Platform` object receives all events at which time all callbacks registered for by the `Platform` objects are executed. Next, all of the callbacks registered for by image objects are executed, then all of the callbacks registered for by album objects are executed.

The name of the event specified in the above call needs to be the fully specified name defined in the GDMO definition unless it is a standard `event` supported by the `Platform` such as `OBJECT_CREATED`, `ATTR_CHANGED` etc. Refer also to the `Platform::get_fullname` method description.

The following function call establishes a callback routine to handle a class of MIS-specific asynchronous events.

For example, the following could be used to find out when an MIS disconnects:

```
when( “DISCONNECTED”, Callback( disconnect_cb, 0));
```

For more information, refer to the “`em_cmipconfig` Example” section in Chapter 4 of the *Solstice Enterprise Manager Application Development Guide* (this book is provided on AnswerBook only).

The following `Platform` events are supported:

Table 3-19 MIS Events Supported

Event	**Description**
ATTR_CHANGED	An attribute of an object in the MIS changed.
`DISCONNECTED`	The MIS dropped its end of the connection for some reason.
OBJECT_CREATED	An object was created in the MIS.
OBJECT_DESTROYE D	An object was destroyed in the MIS.
`RAW_EVENT`	A raw event came in from the MIS. You can examine it before the PMI does anything else with it. But note that `CurrentEvent::do_something` never does anything with a raw event. You have to register a callback for the event under the proper name, such as `ATTR_CHANGED`, and `do_something` in that callback.
`WAIT`	The PMI is entering a wait state. CurrentEvent::`get_name` returns a string describing the wait state. The callback is called again when leaving the wait state, but with a null name.

3.7.13 `Syntax` *Class*

Inheritance: `public Error`

`#include <pmi/hi.hh>`

Data Members: No public data members declared in this class

A `Syntax` is the representation of a type. All framework-encoded data, whether part of an object or not, has a type. This type specifies, among other things, how to produce and understand human-readable representations of the data. In general, the application programmer need not deal with `Syntax`es directly except when building `Morf`s from scratch.

Table 3-20 Syntax Method Types

Method Name	**Method Type**

`expansion`
`get`
`get_raw`
	Access to the type representation

`get_platform`
`is_choice`
`is_list`
	Collateral information

`get_member_names`
`get_memname`
`member`
	List handling

`get_coder` `set_coder`	String encoding / decoding hooks

3.7.13.1 Constructors

```
Syntax()
```

The default constructor creates a `Syntax` instance that refers to no actual `Syntax`. The value tests false until you assign it a real `Syntax` value.

```
Syntax( const Syntax& <other>)
```

The preceding constructor is an ordinary copy constructor. After the copy, both copies still refer to the same `Syntax` object. The reference count on the `Syntax` object is incremented.

```
Syntax( Platform& <plat>, DU <text>)
```

The preceding constructor constructs a `Syntax` instance for a particular kind of MIS. Because a `Syntax` is really a wrapper for a set of related classes, this function actually works somewhat like a virtual constructor.

```
Syntax( Morf& <morf>)
```

The preceding constructor constructs a `Syntax` instance for a particular kind of MIS, which is an implicit part of the `Morf`. The data portion of the `Morf` is interpreted as a description of the correct `Syntax`. (The `Syntax` implicit to the `Morf` describes the syntax of the description, not the `Syntax` to be created by the constructor.) Because a `Syntax` is really a wrapper for a set of related classes, this function actually works a little like a virtual constructor.

```
Syntax(CDU <attrname>, Platform& <plat> = Platform::def_platform)
```

The preceding constructor constructs a `Syntax` instance for a particular kind of attribute. You could conceivably accomplish the same end by creating an image for an object of the type containing the attribute in question, extracting the `Morf` corresponding to that attribute, and then extracting the `Syntax` from that `Morf`. However, this is far more cumbersome than simply instantiating a `Syntax` instance.

3.7.13.2 *Operator Overloading for* `Syntax`

```
Syntax& operator = (const Syntax& <other>)
```

The assignment operator works just like the copy constructor.

```
operator void*()
```

The preceding cast operator is for use in conditionals. It returns true (that is, a nonzero value) if this `Syntax` refers to an actual syntax object. Do not attempt to use the returned value as a pointer to anything, since it points to private data.

```
int operator !()
```

The preceding function call is provided so that you can use “`if (!syntax)`…”.

3.7.13.3 *Member Functions of* `Syntax`

This section describes the member functions of the `Syntax` class.

expansion()

The purpose of the `Syntax::expansion()` is to return the syntax, without respect to tagging or selection, from the current syntax.

get

```
DU get() const
```

```
Coder get_coder()
```

Returns a textual expansion of the syntax.

get_member_names

The preceding function call returns the encoder/decoder for this type.

```
Array(DU) get_member_names()
```

The preceding function call returns the list of member names (field names) for list types that have such names.

get_memname

```
DU get_memname()
```

The preceding function call returns the member name (field name) of this syntax, if it happens to be a member of a list that has member names.

get_platform

```
Platform get_platform()
```

The preceding function call returns the MIS implicit in every `Syntax`.

get_raw

```
Morf get_raw()
```

The preceding function call returns the `Syntax` in encoded, MIS-specific form.

is_choice

```
Boolean is_choice()
```

The preceding function call returns true if the `Syntax` describes a type that lets you pick one of a set of other types.

is_list

```
Boolean is_list()
```

The preceding function call returns true if the `Syntax` describes a compound data value, and false if it describes a scalar value.

is_sequence

```
Boolean is_sequence()
```

The preceding function call returns true if the syntax describes a compound data value that contains an ordered list of zero or more members.

is_set

```
Boolean is_set()
```

The preceding function call returns true if the syntax describes a compound data value that contains an unordered list of zero or more members.

member

```
Syntax member( DU <name> = duNO_VALUE )
```

The preceding function call returns the `Syntax` for a type that is a member of the current type. Obviously, this is valid only for `Syntax`es that have members, namely choices and lists. If a list type has unnamed members (or is a list of identical elements, such as a “SET OF” or “SEQUENCE OF” ASN-1 type) the first anonymous member type can be extracted by passing a null <*name*> argument. (Subsequent anonymous types are inaccessible.)

set_coder

```
void set_coder( Coder* <coder>)
```

The preceding function call sets the encoder / decoder for this type. The `Coder` is used by the `get_str` and `set_str` functions in both `Morf`s and `Image`s. See also `set_attr_coder` under the description of the `Platform` class, which starts on page 3-109.

3.7.14 `Waiter` Class

Inheritance: `public Error`

`#include <pmi/hi.hh>`

Data Members: No public data members declared in this class

A `Waiter` is the representation of an ongoing asynchronous operation. The `Waiter` provides methods for cancelling and awaiting completion of the operation.

The `Waiter` can also serve as the basis for asynchronous operations of your own construction.

Table 3-21 Waiter Method Types

Method Name	**Method Type**
`wait`	Normal Wait

`when_canceled`
`when_done`
`when_tick`
	Schedule additional notifications

`cancel`
`get_except`
`num_clobbered`
`time_remaining`
`waitmore`
`was_completed`
	Managing a Waiter you didn't create

`clobber`
`complete`
`dec`
`get_current_event`
`get_data`
`inc`
`ref`
	Managing a Waiter you created

3.7.14.1 Constructors

```
Waiter()
```

The default constructor creates a `Waiter` instance that refers to no actual `Waiter`. The value tests false until you assign it a real `Waiter` value.

```
Waiter( const Waiter& <other>)
```

The preceding constructor is an ordinary copy constructor. After the copy, both copies still refer to the same `Waiter` object. The reference count on the `Waiter` object is incremented.

```
Waiter( Ptr <ptrdata>,
CCB <callback>,
Timeout <defto> = 60.0)
```

The preceding constructor constructs a `Waiter` instance that calls back to the specified *<callback>* when the `Waiter` completes. The *<ptrdata>* is just a convenient place to store arbitrary data.

```
Waiter( Ptr <ptrdata>, Boolean <reuse> = FALSE )
```

The preceding constructor constructs a `Waiter` instance from a `void*` pointer created by the `ref` method. It is primarily for internal PMI use within callbacks, when the callback can occur after the original `Waiter` has gone out of scope, and would ordinarily have been deleted. Each call to `ref` increments a reference count, and each construction of a `Waiter` using this constructor eventually causes the reference count to be decremented again when the `Waiter` is destructed at the end of the callback. If multiple callbacks are to use the same pointer, then pass a *<reuse>* parameter of true on all but the last callback (or just call `ref` again within the callback) to keep the reference alive till the next callback.

```
Waiter::complete
```

After the preceding constructor is constructed from a `void*` pointer by the `ref()` method, make sure the `waiter::complete()` function is called. An example is shown below.

Code Example 3-2 `waiter::complete()` Function

```
    Callback cb_all(all_done, 0);
    Waiter all_waiter(0, cb_all);

    cout << "Image(" << obj1 << ") to be booted" << endl;
    Image im1 = Image(obj1);
    Callback cb1(f1, all_waiter.ref());
    Waiter waiter1 = im1.start_boot(cb1);

    cout << "Image(" << obj2 << ") to be booted" << endl;
    Image im2 = Image(obj2);
    Callback cb2(f2, all_waiter.ref());
    Waiter waiter2 = im2.start_boot(cb2);

    cout << "Waiting for callback event" << endl;

    while ( !done_flag ) {
        dispatch_recursive(TRUE);
    }


void
f1(Ptr user_data, Ptr)
{
        cout << "job1 is done" << endl;

        Waiter waiter1(user_data, FALSE);

        if (waiter1)
        {
                //Mark job1 is done
                waiter1.complete();
        }
}
void
f2(Ptr user_data, Ptr)
{
        cout << "job2 is done" << endl;

        Waiter waiter2(user_data, FALSE);
```

Code Example 3-2 `waiter::complete()` Function

```
        if (waiter2)
        {
                //Mark job2 is done
                waiter2.complete();
        }
}

void
all_done(Ptr, Ptr)
{
        cout << "All Jobs are done" << endl;

        done_flag = 1;
}
```

3.7.14.2 *Operator Overloading for Waiter*

```
Waiter& operator = ( const Waiter& <other>)
```

The assignment operator, above, works just like the copy constructor.

```
operator void*()
```

The preceding cast operator is for use in conditionals. It returns true if this `Waiter` refers to an actual `Waiter` object. Do not attempt to use the returned value as a pointer to anything, since it points to private data.

```
int operator !()
```

The preceding function call is provided so that you can say "`if (!waiter)`…"

3.7.14.3 *Member Functions of Waiter*

This section describes the member functions of the `Waiter` class.

cancel

```
Result cancel()
```

The preceding function call causes the asynchronous operation to time out immediately. Essentially it's just a `waitmore(0.0)`.

clobber

```
void clobber( const ExceptionType* <err> = 0)
```

The preceding function call causes the `Waiter` to be marked as deficient in some respect or other. This function is primarily for internal use; the PMI uses it to pass error information back to the application when, for instance, an unexpected error response is received from the MIS. The function can be called multiple times, but only the first error is remembered. This function does not complete the `Waiter`.

complete

```
void complete()
```

The preceding function call marks the `Waiter` as complete. This function is primarily for internal use; the PMI uses it to notify the `Waiter` so that it can call any waiting external callbacks, and can let the `wait` function return to the application (if the `wait` function was in fact called). Waiter.complete() should be called regularly. Calls to this function are ignored if there are still pending internal callbacks.

dec

```
U32 dec()
```

The preceding function call decrements the `Waiter`'s count of the number of internal callbacks it is waiting for. The `Waiter` cannot complete while there are pending internal callbacks. This function is primarily for internal use; it returns the number of callbacks pending after the decrement.

get_current_event

```
CurrentEvent get_current_event()
```

The preceding function call returns the `CurrentEvent` that is contained within the `Waiter`. This is the event that is passed to all external callbacks when the `Waiter` completes. The "message pointer" in this event is actually a pointer back to the `Waiter` containing it.

get_data

```
Ptr get_data()
```

The preceding function call returns the <*ptrdata*> originally passed to the `Waiter` constructor. This function is primarily for internal use.

get_except

```
ExceptionType* get_except()
```

The preceding function call returns the exception that the `Waiter` was clobbered with, if any.

inc

```
U32 inc()
```

The preceding function call increments the `Waiter`'s count of the number of internal callbacks it is waiting for. The `Waiter` cannot complete while there are still pending internal callbacks. This function is primarily for internal use. It returns the number pending before the increment.

Note – It can be disastrous to lose track of the number of pending callbacks. You generally either hang forever or dump core.

num_clobbered

```
U32 num_clobbered()
```

The preceding function call returns the number of times the `Waiter` was clobbered.

ref

```
Ptr ref()
```

The preceding function call returns a reference-counted `void*` pointer to this `Waiter`, from which you *must*, at some future time, reconstruct the `Waiter` using the `Waiter(Ptr, Boolean)` constructor. Refer to the description of that constructor earlier in this section for further information.

Note – If Waiter::ref() is used to pass waiters in callbacks, then the asynchronous operations never completes. This is because the ref() function may do more than just increase the ref count on the Waiter. It also increases the pending count on the Waiter.

time_remaining

```
Timeout time_remaining()
```

The preceding function call returns the time remaining before the `Waiter` would expire due to a time-out. In combination with the `when_tick` function, this let's you give the user a countdown till the time the application blows up. Note that a `Timeout` value is a (possibly fractional) number of seconds. Any rounding is up to you.

wait

```
Result wait( Timeout <to> = DEFAULT_TIMEOUT)
```

The preceding function call blocks for up to the specified period, waiting for the asynchronous operation to complete. It returns a true value if the operation completes. It it also capable of throwing an exception if the `Waiter` was clobbered with a nonzero `ExceptionType` pointer.

waitmore

```
Result waitmore( Timeout <to> = DEFAULT_TIMEOUT)
```

The preceding function call is available for callbacks to reset the time-out clock to a longer (or shorter) interval because some intermediate event occurred. For instance, if receiving multiple messages, you might want to extend the time-out each time a message comes in, and only time out if the gap between two subsequent messages exceeds some threshold.

was_completed

```
Boolean was_completed()
```

The preceding function call returns true if the `Waiter::complete` function has been called when there were no more pending internal callbacks.

when_cancelled

```
Result when_cancelled( CCB <cb> = NO_CALLBACK)
```

The preceding function call specifies a callback to call if the operation is cancelled or times out or is otherwise clobbered. Multiple callbacks can be added per waiter. If the callback is not specified, all callbacks are removed, *including* the original callback passed to the constructor.

when_done

```
Result when_done( CCB <cb> = NO_CALLBACK)
```

The preceding function call specifies a callback to call if the operation completes successfully. Multiple callbacks can be added per waiter. If the callback is not specified, all callbacks are removed, *including* the original callback passed to the constructor.

when_tick

```
Result when_tick( CCB <cb> = NO_CALLBACK, Timeout <to> = 1.0)
```

The preceding function call specifies a callback to call repeatedly as long as the operation has not yet completed successfully. By default the callback is called once per second, but you can specify a different `Timeout` parameter to modify that. Only one such callback can be added per waiter. If the callback is not specified, the callback is removed.

Low-Level PMI 4

4.1 Introduction

The Solstice EM Portable Management Interface (PMI) provides a low-level CMIS-like, distributed, transport-independent interface for application programs into the Message Routing Module (MRM)

The low-level PMI uses paired sets of Transport-Independent and Transport-Dependent SAPs to provide a communication path between an application and the MRM. The set of paired SAPs use the transport mechanism specified by the transport dependent SAP to pass messages between SAPs (between an application and the MRM). Each set of SAPs normally resides in separate Unix processes. Solstice EM currently uses a CMIS-like protocol over the Lightweight Presentation Protocol (LPP) and TCP/IP to pass messages between the paired SAPs.

Figure 4-1 shows how the low-level PMI communicates between applications and the MRM.

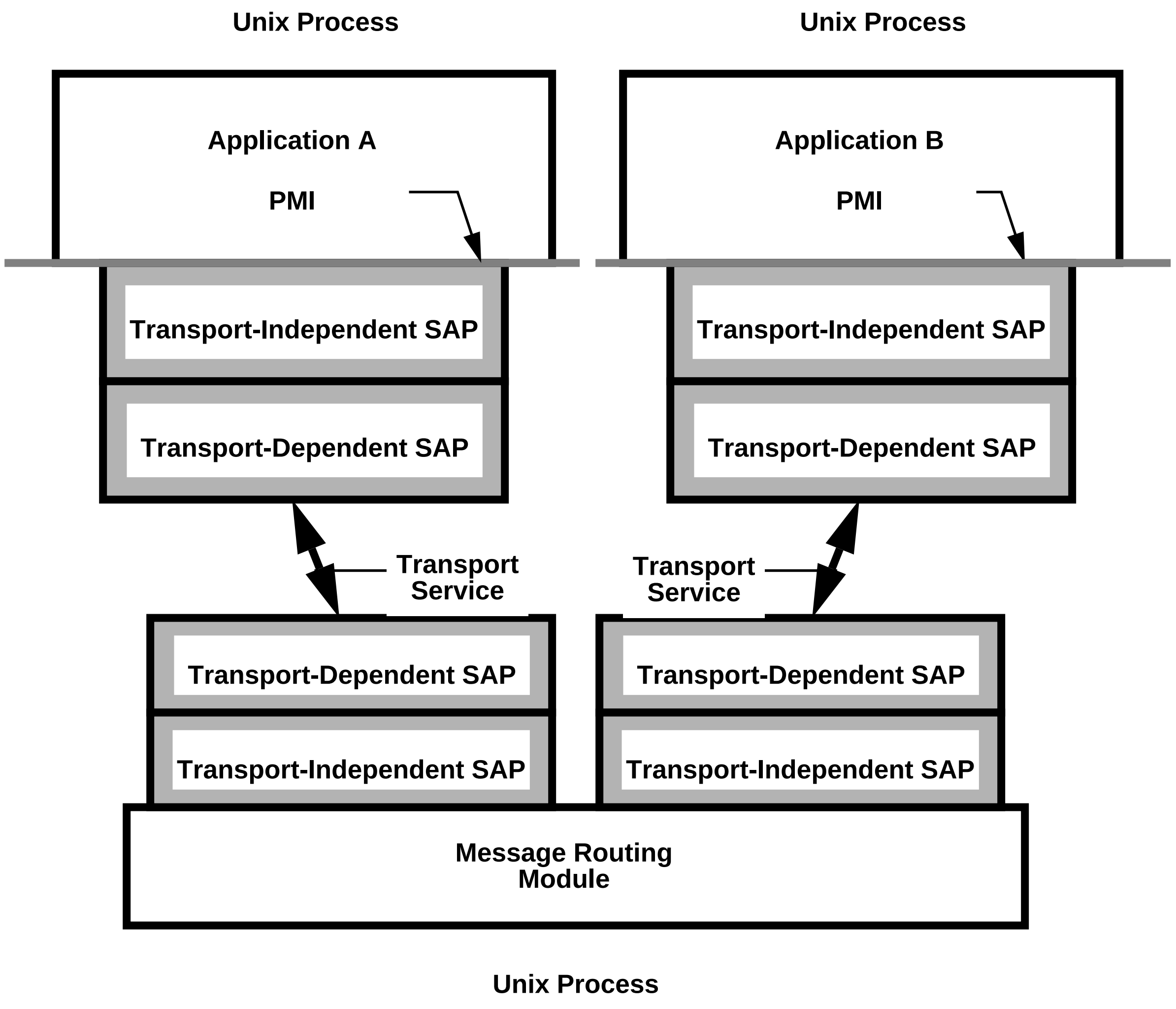


Figure 4-1 Applications to MRM Communication

4.2 Root Classes for the Low Level PMI

The root classes for the low level PMI include:

- *Message Class*
- *MessageSAP Class*
- *MessScope Class*

In addition, the Asn1Value and DataUnit classes (Chapter 2, “Common API Classes”) are common base classes.

The data contained in the class structures based on `Message` class and defined in the `/opt/SUNWconn/em/include/pmi/message.h` file are primarily based on the `Asn1Value` class, defined in `asn1_val.hh`. The `Asn1Value` class in turn relies on structures and methods defined in the `DataUnit` class, defined in `du.hh`

4.3 Low-Level PMI Classes

4.3.1 Class Summary

The low-level PMI includes the following classes:

Table 4-1 Low-Level PMI Classes

Class	Description
AccessDenied Class	Adds an Asn1Value parameter to store a current time
ActionReq Class	Serves as a repository for information identifying an action request message
ActionRes Class	Add three Asn1Value parameters to store information
AssocReleased Class	Represents an error message
CancelGetReq Class	Adds inherited functions/variables and a MessID parameter
CancelGetRes Class	Contains member functions/variables of its derived classes
ClassInstConfl Class	Adds parameters to store a class and instance
CreateReq Class	Add parameters to store s superior object class
CreateRes Class	Add parameters to store a current time an attribute list

Table 4-1 Low-Level PMI Classes

Class	**Description**
DeleteReq Class	Contains member functions/variables of a derived class
DeleteRes Class	Adds parameter to store a current time
DuplicateOI Class	Adds an object instance parameter
DupMessageId Class	Contains member functions/variables of a derived class
ErrorResUnexp Class	Contains member functions/variables of a derived class
EventReq Class	Adds parameters to store an event type, time and information
GetListErr Class	Add parameters to store a current time and get an information list
GetReq Class	Adds an attribute ID list parameter
GetRes Class	Adds parameters to store a current time and an attribute list
InvalidActionArg Class	Adds parameters to store a current time and some action information
InvalidAttrVal Class	Add an attribute parameter
InvalidEventArg Class	Adds parameters to store an object class, event type, and some event information
InvalidFilter Class	Adds a filter parameter
InvalidOI Class	Add an object instance parameter
InvalidOperation Class	Contains member functions/variables of a derived class
InvalidOperator Class	Adds a modified operator parameter
InvalidScope Class	Adds a scope parameter
LinkedResUnexp Class	Contains member functions/variables of derived class
Message Class	Contains data that is common to every type of message sent via a MessageSAP interface
MessageSAP Class	Defines queues of pointers to messages
MessQOS Class	Represents the Quality of Service indicator included in all messages
MessScope Class	Defines a message's scope - the range of objects where a message is applied

Table 4-1 Low-Level PMI Classes

Class	**Description**
MissingAttrVal Class	Adds an attribute ID list parameter
MistypedArg Class	Contains member functions/variables of a derived class
MistypedError Class	Contains member functions/variables of a derived class
MistypedOp Class	Contains member functions/variables of a derived class
MistypedRes Class	Contains member functions/variables of a derived class
NoSuchAction Class	Adds parameters to store a current time and an action type
NoSuchActionArg Class	Adds parameters to store a current time and an action type
NoSuchAttr Class	Adds an attribute ID parameter
NoSuchEvent Class	Adds parameters to store an object class and an event type
NoSuchEventArg Class	Adds parameters to store an object class and an event type
NoSuchMessageId Class	Adds a get ID parameter
NoSuchOC Class	Adds a parameter to store an object class
NoSuchOI Class	Adds a parameter to store an object instance
NoSuchRefOI Class	Adds an object instance parameter
ObjReqMess Class	Adds parameters for an object class and instance
ObjResMess Class	Adds parameters for an object class and instance
OpCancelled Class	Contains member functions/variables of derived class
ProcessFailure Class	Adds a specific error information parameter
ReqMess Class	Adds a parameter to a message
ResMess Class	Adds a variable that indicates a linked response message
ResourceLimit Class	Contains member functions/variables of a derived class
ScopedReqMess Class	Adds variables and parameters
SetListErr Class	Adds parameters to store a current time and a set information list
SetReq Class	Contains member functions/variables of a derived class

Table 4-1 Low-Level PMI Classes

Class	Description
SetRes Class	Adds parameters to store a current time and an attribute list
SyncNotSupp Class	Adds a sync parameter
TimedOut Class	Generates an error message
UnexpChildOp Class	Contains member functions / variables of a derived class
UnexpError Class	Represents an error message
UnexpRes Class	Represents an error message
UnrecError Class	Represents an error message
UnrecLinkedId Class	Represents an error message
UnrecMessageId Class	Represents an error message
UnrecOp Class	Represents an error message

4.3.2 `AccessDenied` *Class*

Inheritance: `class AccessDenied : public ObjResMess, public ResMess, public Message, public QueueElem`

`#include <pmi/message.hh>`

Method types: No public member functions are declared in this class.

In addition to the functions or variables it inherits, the `AccessDenied` class adds an Asn1Value parameter to store a current time. The usage of this parameter is described in detail in the CMIS documentation covering the `accessDenied` error. The `oc`, `oi`, and `curr_time` members are only defined when returning an error from a scoped `ACTION_REQ` or `DELETE_REQ`.

Table 4-2 lists the `AccessDenied` public variable:

Table 4-2 AccessDenied Public Variable

Type	Variable	Description
Asn1Value	`curr_time;`	An optional parameter specifying the time that this response message was generated.

4.3.2.1 *Constructor*

```
AccessDenied()
```

The constructor for `AccessDenied` takes no parameters. It initializes its parent class.

4.3.3 `ActionReq` *Class*

Inheritance: `class ActionReq : public ScopedReqMess, public ObjReqMess, public ReqMess, public Message`

`#include <pmi/message.hh>`

Method types: No public member functions are declared in this class.

An instance of `ActionReq` serves as a repository for information identifying an action request message.

Table 4-3 lists the `ActionReq` public data members:

Table 4-3 ActionReq Public Data Members

Type	**Variables**	**Description**
Asn1Value	`action_type;`	The type of action being requested by this message.
Asn1Value	`action_info;`	Information that might be included in this action request. The data content of this parameter depends on the `action_type`. There are definitions for the action types in the OSI Network Management Forum document.

4.3.3.1 *Constructor*

```
ActionReq()
```

The constructor for `ActionReq` takes no parameters and does nothing more than initialize its parent class(es).

4.3.4 `ActionRes` *Class*

Inheritance: `class ActionRes : public ObjResMess, public ResMess, public Message, public QueueElem`

`#include <pmi/message.hh>`

Method Types: No public member functions are declared in this class.

In addition to the functions or variables it inherits, the `ActionRes` class adds three Asn1Value parameters to store a current time, an action type, and some action reply info. The usage of these parameters is described in detail in the CMIS documentation covering the `Action` response.

Table 4-4 lists the `ActionRes` public data members:

Table 4-4 ActionRes Public Data Members

Type	Variables	Description
Asn1Value	`curr_time;`	An optional parameter specifying the time that this response message was generated.
Asn1Value	`action_type;`	The type of action for which this response is being generated.
Asn1Value	`action_reply;`	Information accompanying this action response. The contents of this optional parameter vary and are based on the action type specified. The formats for this parameter for the various action types are given in the GDMO and ASN.1 documents.

4.3.4.1 *Constructor*

```
ActionRes()
```

The constructor for `ActionRes` takes no parameters. It only initializes its parent class(es).

4.3.5 `AssocReleased` *Class*

Inheritance: `class AssocReleased : public ResMess, public Message, public QueueElem`

`#include <pmi/message.hh>`

Method Types: No public member functions are declared in this class.

The `AssocReleased` class defines no functions or variables beyond those it inherits. The error message that this class represents is patterned after the ROSE error message initiator-releasing. The usage of this message is described in detail in the documentation covering the ROSE protocol. It is used within the Solstice EM MIS to indicate that a request could not be serviced because the association on which that request was received is either about to or has already gone away.

4.3.5.1 Constructor

```
AssocReleased()
```

This constructor takes no parameters. It initializes its parent class(es).

4.3.6 `CancelGetReq` *Class*

Inheritance: `class CancelGetReq : public ReqMess, public Message, public QueueElem`

`#include <pmi/message.hh>`

Method Types: No public member functions are declared in this class.

In addition to the functions or variables it inherits, the `CancelGetReq` class adds a `MessId` parameter. This parameter specifies the ID of the CMIS Get request that is being cancelled by this request. The use of this message is described in the CMIS documentation covering the `CancelGet` request.

Table 4-5 lists the `CancelGetReq` public variable:

Table 4-5 CancelGetReq Public Variable

Type	**Variable**	**Description**
MessId	get_id;	The id of the CMIS Get request that is being cancelled by this request.

4.3.6.1 Constructor

```
CancelGetReq()
```

This constructor takes no parameters. It only initializes its parent class(es).

4.3.7 `CancelGetRes` *Class*

Inheritance: `class CancelGetRes : public ResMess, public Message, public QueueElem`

`#include <pmi/message.hh>`

Method Types: No public member functions are declared in this class.

The `CancelGetRes` object class contains all of the member variables and member functions that are present in the classes it has derived from, whether directly or indirectly. No additional parameters are available for this response message. The use of this message is described in detail in the CMIS documentation covering the `CancelGet` response.

4.3.7.1 Constructor

```
CancelGetRes()
```

This constructor takes no parameters. It only initializes its parent class(es).

4.3.8 `ClassInstConfl` *Class*

Inheritance: `class ClassInstConfl : public ResMess, public Message, public QueueElem`

`#include <pmi/message.hh>`

Method Types: No public member functions are declared in this class.

In addition to the functions or variables it inherits, the `ClassInstConfl` class adds two Asn1Value parameters to store an object class and an object instance. The usage of these parameters is described in detail in the CMIS documentation covering the `classInstanceConflict` error.

Table 4-6 lists the `ClassInstConfl` public data members:

Table 4-6 ClassInstConfl Public Data Members

Type	Variables	Description
Asn1Value	`oc;`	The object class that was specified in the request message.
Asn1Value	`oi;`	The object instance, whose object class is not the same as `oc`, that caused the generation of this error message.

4.3.8.1 *Constructor*

```
ClassInstConfl()
```

This constructor takes no parameters. It only initializes its parent class(es).

4.3.9 `CreateReq` *Class*

Inheritance: `class CreateReq : public ObjReqMess, public ReqMess, public Message, public QueueElem`

`#include <pmi/message.hh>`

Method Types: No public member functions are declared in this class.

In addition to the functions or variables it inherits, the `CreateReq` class adds four Asn1Value parameters to store a superior object instance, access control information, a reference object instance, and an attribute list. The usage of these parameters is described in detail in the CMIS documentation covering the Create request.

Table 4-7 lists the `CreateReq` public data members:

Table 4-7 CreateReq Public Data Members

Type	**Variables**	**Description**
Asn1Value	`superior_oi;`	The object that will be the parent object (in the MIT) to this newly created object.
Asn1Value	`access;`	Access control information that is checked by the destination to determine if the issuer of this request message is allowed to perform a creation. This parameter is optional.
Asn1Value	`reference_oi;`	The optional object instance of an object whose attributes are to be copied into this new object.
Asn1Value	`attr_list;`	An optional list of attributes that the newly created object is to contain.

4.3.9.1 Constructor

```
CreateReq()
```

This constructor takes no parameters. It initializes its parent class(es) and its internal data.

4.3.10 `CreateRes` *Class*

Inheritance: `class CreateRes : public ObjResMess, public Message, public QueueElem`

`#include <pmi/message.hh>`

Method Types: No public member functions are declared in this class.

In addition to the functions or variables it inherits, the `CreateRes` class adds two Asn1Value parameters to store a current time and an attribute list. The usage of these parameters is described in detail in the CMIS documentation covering the Create response.

Table 4-8 lists the `CreateRes` public data members:

Table 4-8 CreateRes Public Data Members

Type	**Variables**	**Description**
Asn1Value	`curr_time;`	An optional parameter specifying the time that this response message was generated.
Asn1Value	`attr_list;`	This optional parameter contains a list of attribute ids and values with which the new object was created.

4.3.10.1 Constructor

```
CreateRes()
```

This constructor takes no parameters. It initializes its parent class(es).

4.3.11 `DeleteReq` *Class*

Inheritance: `class DeleteReq: public ScopedReqMess, public ObjReqMess, public ReqMess, public Message, public QueueElem`

`#include <pmi/message.hh>`

Method Types: No public member functions are declared in this class.

The `DeleteReq` object class contains all of the member variables and member functions that are present in the classes that it has derived from, either directly or indirectly. It does not add any parameters other that those that it inherits. The use of this message is described in the CMIS documentation covering the Delete request.

4.3.11.1 Constructor

```
DeleteReq()
```

This constructor takes no parameters. It only initializes its parent class(es).

4.3.12 `DeleteRes` *Class*

Inheritance: `class DeleteRes : public ObjResMess, public ResMess, public Message, public QueueElem`

`#include <pmi/message.hh>`

Method Types: No public member functions are declared in this class.

In addition to the functions or variables it inherits, the `DeleteRes` class adds an Asn1Value parameter to store a current time. The usage of this parameter is described in detail in the CMIS documentation covering the Delete response.

Table 4-9 lists the `DeleteRes` public variable:

Table 4-9 DeleteRes Public Variable

Type	**Variable**	**Description**
Asn1Value	`curr_time;`	An optional parameter specifying the time that this response message was generated.

4.3.12.1 Constructor

```
DeleteRes()
```

This constructor takes no parameters. It only initializes its parent class(es).

4.3.13 `DuplicateOI` *Class*

Inheritance: `class Duplicate : public ResMess, public Message, public QueueElem`

`#include <pmi/message.hh>`

Method Types: No public member functions are declared in this class.

In addition to the functions or variables it inherits, the `DuplicateOI` class adds an object instance parameter. The usage of this parameter is described in detail in the CMIS documentation covering the `duplicateManagedObjectInstance` error.

Table 4-10 lists the `DuplicateOI` public variable:

Table 4-10 DuplicateOI Public Variable

Type	**Variable**	**Description**
Asn1Value	`oi;`	This is an invalid object instance that caused the generation of this error message. This is sent in response to a create request when the object whose creation was requested was given the same object instance as an already existing object.

4.3.13.1 Constructor

```
DuplicateOI()
```

This constructor takes no parameters. It only initializes its parent class(es).

4.3.14 `DupMessageId` *Class*

Inheritance: `class DupMessageId : public ResMess, class Message, class QueueElem`

`#include <pmi/message.hh>`

Data Members: No public data members declared in this class.

Method Types: No public member functions are declared in this class.

The `DupMessageId` object class contains all of the member variables and member functions that are present in the classes that it has derived from, either directly or indirectly. This class defines no functions or variables beyond those it inherits. The error message that this class represents is patterned after the ROSE error message duplicate-invocation. The usage of this message is described in detail in the documentation covering the ROSE protocol. It is used within the Solstice EM MIS to indicate that a module/application generated a request message with the same message id as an already existing and outstanding request.

4.3.14.1 Constructor

```
DupMessageId()
```

This constructor takes no parameters. It only initializes its parent class(es).

4.3.15 `ErrorResUnexp` *Class*

Inheritance: `class ErrorResUnexp : public ResMess, public Message, public QueueElem`

`#include <pmi/message.hh>`

Data Members: No public data members declared in this class.

Method Types: No public member functions are declared in this class.

The `ErrorResUnexp` object class contains all of the member variables and member functions that are present in the classes that it has derived from, either directly or indirectly. This class defines no functions or variables beyond those it inherits. The error message that this class represents is patterned after the ROSE error message error-response-unexpected. It is used within the Solstice EM MIS to indicate that an error message was generated in response to a non-confirmed request.

4.3.15.1 Constructor

```
ErrorResUnexp()
```

This constructor takes no parameters. It only initializes its parent class(es).

4.3.16 `EventReq` *Class*

Inheritance: `class EventReq : public ObjReqMess, public ReqMess, public Message, public QueueElem`

`#include <pmi/message.hh>`

Method Types: No public member functions are declared in this class.

In addition to the functions or variables it inherits, the `EventReq` class adds three Asn1Value parameters to store an event type, an event time, and some event information. The usage of these parameters is described in detail in the CMIS documentation covering the `EventReport` request message.

Table 4-11 lists the `EventReq` public data members:

Table 4-11 EventReq Public Data Members

Type	**Variables**	**Description**
Asn1Value	`event_type;`	The type of event reported via this message.
Asn1Value	`event_time;`	The time that the originator of the event chose to place here (probably but not necessarily the time that the event occurred).
Asn1Value	`event_info;`	Any supplemental information that is to accompany this request. Specific data formats for this parameter depend on the <*event_type*> and are defined in OSI Network Management Forum documentation.

4.3.16.1 *Constructor*

```
EventReq()
```

This constructor takes no parameters. It only initializes its parent class(es).

4.3.17 `GetListErr` Class

Inheritance: `class GetListErr : public ObjResMess, public Message, public QueueElem`

`#include <pmi/message.hh>`

Method Types: No public member functions are declared in this class.

In addition to the functions or variables it inherits, the `GetListErr` class adds two Asn1Value parameters to store a current time and a get an information list. The usage of these parameters is described in detail in the CMIS documentation covering the `getListError` error.

Table 4-12 lists the `GetListErr` public data members:

Table 4-12 GetListErr Public Data Members

Type	**Variables**	**Description**
Asn1Value	`get_info_list;`	The list of attributes that were requested in the Get request. This list contains any attributes which could be accessed. Those attributes which could not be accessed, and hence caused the generation of this error message, are not included in this list.
Asn1Value	`curr_time;`	An optional parameter specifying the time that this response message was generated.

4.3.17.1 Constructor

```
GetListErr()
```

This constructor takes no parameters. It only initializes its parent class(es).

4.3.18 `GetReq` *Class*

Inheritance: `class GetReq : public ScopedReqMess, public ObjReqMess, public ReqMess, public Message, public QueueElem`

`#include <pmi/message.hh>`

Method Types: No public member functions are declared in this class.

In addition to the functions or variables it inherits, the `GetReq` class adds an attribute ID list parameter. The usage of this parameter is described in detail in the CMIS documentation covering the Get request message.

Table 4-13 lists the `GetReq` public variable:

Table 4-13 GetReq Public Variable

Type	**Variable**	**Description**
Asn1Value	`attr_id_list;`	A list of attribute ids whose attribute values are to be returned in the response to this Get request.

4.3.18.1 *Constructor*

```
GetReq()
```

This constructor takes no parameters. It only initializes its parent class(es).

4.3.19 `GetRes` *Class*

Inheritance: `class GetRes : public ObjResMess, public ResMess, public Message, public QueueElem`

`#include <pmi/message.hh>`

Method Types: No public member functions are declared in this class.

In addition to the functions or variables it inherits, the `GetRes` class adds two ASN1Value parameters to store a current time and an attribute list. The usage of these parameters is described in detail in the CMIS documentation covering the Get response.

Table 4-14 lists the `GetRes` public data members:

Table 4-14 GetRes Public Data Members

Type	**Variables**	**Description**
Asn1Value	`curr_time`;	An optional parameter specifying the time that this response message was generated.
Asn1Value	`attr_list;`	A list of attributes was specified in the Get request message for which this response is being generated. This parameter represents the list of attributes that were compiled and are being returned to the requestor.

4.3.19.1 Constructor

```
GetRes()
```

This constructor takes no parameters. It only initializes its parent class(es).

4.3.20 `InvalidActionArg` *Class*

Inheritance: `class InvalidActionArg : public ResMess, public Message, public QueueElem`

`#include <pmi/message.hh>`

Method Types: No public member functions are declared in this class.

In addition to the functions or variables it inherits, the `InvalidActionArg` class adds two Asn1Value parameters to store a current time and some action information. The usage of these parameters is described in detail in the CMIS documentation covering the `invalidArgument` error. There are two choices defined for the `invalidArgument` error message and this class defines the `actionValue` choice (first of the two). The `oi` and `curr_time` members are only defined when returning an error from a scoped `ACTION_REQ`.

Table 4-15 lists the `InvalidActionArg` public data members:

Table 4-15 InvalidActionArg Public Data Members

Type	**Variables**	**Description**
Asn1Value	`action_info;`	Additional information about the action, revealing why this error message was generated.
Asn1Value	`curr_time;`	An optional parameter specifying the time that this response message was generated.

4.3.20.1 Constructor

```
InvalidActionArg()
```

This constructor takes no parameters. It only initializes its parent class(es).

4.3.21 `InvalidAttrVal` *Class*

Inheritance: `class InvalidAttrVal : public ResMess, public Message, public QueueElem`

`#include <pmi/message.hh>`

Method Types: No public member functions are declared in this class.

In addition to the functions or variables it inherits, the `InvalidAttrVal` class adds an attribute parameter. The usage of this parameter is described in detail in the CMIS documentation covering the `invalidAttributeValue` error.

Table 4-16 lists the `InvalidAttrVal` public variable:

Table 4-16 InvalidAttrVal Public Variable

Type	**Variable**	**Description**
Asn1Value	`attr;`	This is the invalid attribute that caused the generation of this error message.

4.3.21.1 Constructor

```
InvalidAttrVal()
```

This constructor takes no parameters. It only initializes its parent class(es).

4.3.22 `InvalidEventArg` *Class*

Inheritance: `class InvalidEventArg : public ResMess, public Message, public QueueElem`

`#include <pmi/message.hh>`

Method Types: No public member functions are declared in this class.

In addition to the functions or variables it inherits, the `InvalidEventArg` class adds three Asn1Value parameters to store an object class, event type, and some event information. The usage of these parameters is described in detail in the CMIS documentation covering the `invalidArgument` error. There are two choices defined for the `invalidArgument` error message and this class defines the `eventValue` choice (second of the two).

Table 4-17 lists the `InvalidEventArg` public data members:

Table 4-17 InvalidEventArg Public Data Members

Type	**Function**	**Description**
Asn1Value	`oc;`	The object that the event report request was generated for.
Asn1Value	`event_type;`	The invalid event type that caused the generation of this error message.
Asn1Value	`event_info;`	Additional information indicating why this error message was generated. The format of this parameter is variable and depends on the <*event_type*> specified in this message.

4.3.22.1 Constructor

```
InvalidEventArg()
```

This constructor takes no parameters. It only initializes its parent class(es).

4.3.23 `InvalidFilter` *Class*

Inheritance: `class InvalidFilter : public ResMess, public Message, public QueueElem`

`#include <pmi/message.hh>`

Method Types: No public member functions are declared in this class.

In addition to the functions or variables it inherits, the `InvalidFilter` class adds a filter parameter. The usage of this parameter is described in detail in the CMIS documentation covering the `invalidFilter` error.

Table 4-18 lists the `InvalidFilter` public variable:

Table 4-18 InvalidFilter Public Variable

Type	**Variable**	**Description**
Asn1Value	`filter;`	The filter that was invalid and caused the generation of this error message.

4.3.23.1 Constructor

```
InvalidFilter()
```

This constructor takes no parameters. It only initializes its parent class(es).

4.3.24 `InvalidOI` *Class*

Inheritance: `class InvalidOI : public ResMess, public Message, public QueueElem`

`#include <pmi/message.hh>`

Method Types: No public member functions are declared in this class.

In addition to the functions or variables it inherits, the `InvalidOI` class adds an object instance parameter. The usage of this parameter is described in detail in the CMIS documentation covering the `invalidObjectInstance` error.

Table 4-19 lists the `InvalidOI` public variable:

Table 4-19 InvalidOI Public Variable

Type	**Variable**	**Description**
Asn1Value	`oi;`	The invalid object instance that caused the generation of this error message.

4.3.24.1 *Constructor*

```
InvalidOI()
```

This constructor takes no parameters. It only initializes its parent class(es).

4.3.25 `InvalidOperation` *Class*

Inheritance: `class InvalidOperation : public ResMess, public Message, public QueueElem`

`#include <pmi/message.hh>`

Data Members: No public data members declared in this class.

Method Types: No public member functions are declared in this class.

The `InvalidOperation` object class contains all of the member variables and member functions that are present in the classes that it has derived from, either directly or indirectly. This class defines no functions or variables beyond those it inherits. The usage of this message is described in detail in the CMIS documentation covering the `invalidOperation` error.

4.3.25.1 *Constructor*

```
InvalidOperation()
```

This constructor takes no parameters. It only initializes its parent class(es).

4.3.26 `InvalidOperator` *Class*

Inheritance: `class InvalidOperator : public ResMess, public Message, public QueueElem`

`#include <pmi/message.hh>`

Method Types: No public member functions are declared in this class.

In addition to the functions or variables it inherits, the `InvalidOperator` class adds a modify operator parameter. The usage of this parameter is described in detail in the CMIS documentation covering the `invalidOperator` error.

Table 4-20 lists the `InvalidOperator` public variable:

Table 4-20 InvalidOperator Public Variable

Type	**Variable**	**Description**
Asn1Value	`mod_op;`	The invalid modify operator that was specified in a Set request and thus caused the generation of this error message.

4.3.26.1 Constructor

```
InvalidOperator()
```

This constructor takes no parameters. It only initializes its parent class(es).

4.3.27 `InvalidScope` *Class*

Inheritance: `class InvalidScope : public ResMess, public Message, public QueueElem`

`#include <pmi/message.hh>`

Method Types: No public member functions are declared in this class.

In addition to the functions or variables it inherits, the `InvalidScope` class adds a scope parameter. The usage of this parameter is described in detail in the CMIS documentation covering the `invalidScope` error.

Table 4-21 lists the `InvalidScope` public variable:

Table 4-21 InvalidScope Public Variable

Type	**Variable**	**Description**
MessScope	`scope;`	The invalid scope parameter, extracted from the request message, which caused the generation of this error message.

4.3.27.1 Constructor

```
InvalidScope()
```

This constructor takes no parameters. It only initializes its parent class(es).

4.3.28 `LinkedResUnexp` *Class*

Inheritance: `class LinkedResUnexp : public ResMess, public Message, public QueueElem`

`#include <pmi/message.hh>`

Data Members: No public data members declared in this class.

Method Types: No public member functions are declared in this class.

The `LinkedResUnexp` object class contains all of the member variables and member functions that are present in the classes that it has derived from, either directly or indirectly. This class defines no functions or variables beyond those it inherits. The error message that this class represents is patterned after the ROSE error message “linked-response-unexpected.” It is used within the Solstice EM MIS to indicate that a linked request was received and the linked ID specified did not refer to a request for which a linked ID could be generated.

4.3.28.1 Constructor

```
LinkedResUnexp()
```

This constructor takes no parameters. It only initializes its parent class(es).

4.3.29 `Message` Class

Inheritance: `class Message:public QueueElem`

`#include <pmi/message.hh>`

The `Message` class is the base class used by almost all messages passed between SAPs and the PMI. The messages contained in the `message.hh` file largely define the syntax of the low-level usage of the PMI.

A number of CMIS-like messages are subclasses of the `Message` class. The three primary types of messages are:

- Request messages
- Response messages
- Error response messages

Solstice EM CMIS messages are derived from `Message`, the base message class. It contains data that is common to every type of message sent via a `MessageSAP` interface. All of the other base message classes derive either directly or indirectly from the `Message` class. You should never instantiate this class, only derive other classes from it.

Other base message classes contain a parameter or set of parameters that are commonly used together in more than one Solstice EM CMIS message. Some messages commonly use more than one set of parameters and therefore, some of the base message classes are combinations of other base message classes formed via inheritance. The parameters that are included in the base message classes include only those parameters that are used by the Message Routing Module (MRM) to perform scoping, filtering, access control, and synchronization.

Note – Each of the classes derived from the `Message` class relies on the ISO specifications of the CMIP protocol and ASN.1 data encoding.

The following diagram shows the inheritance hierarchy for the classes based on `Message`:

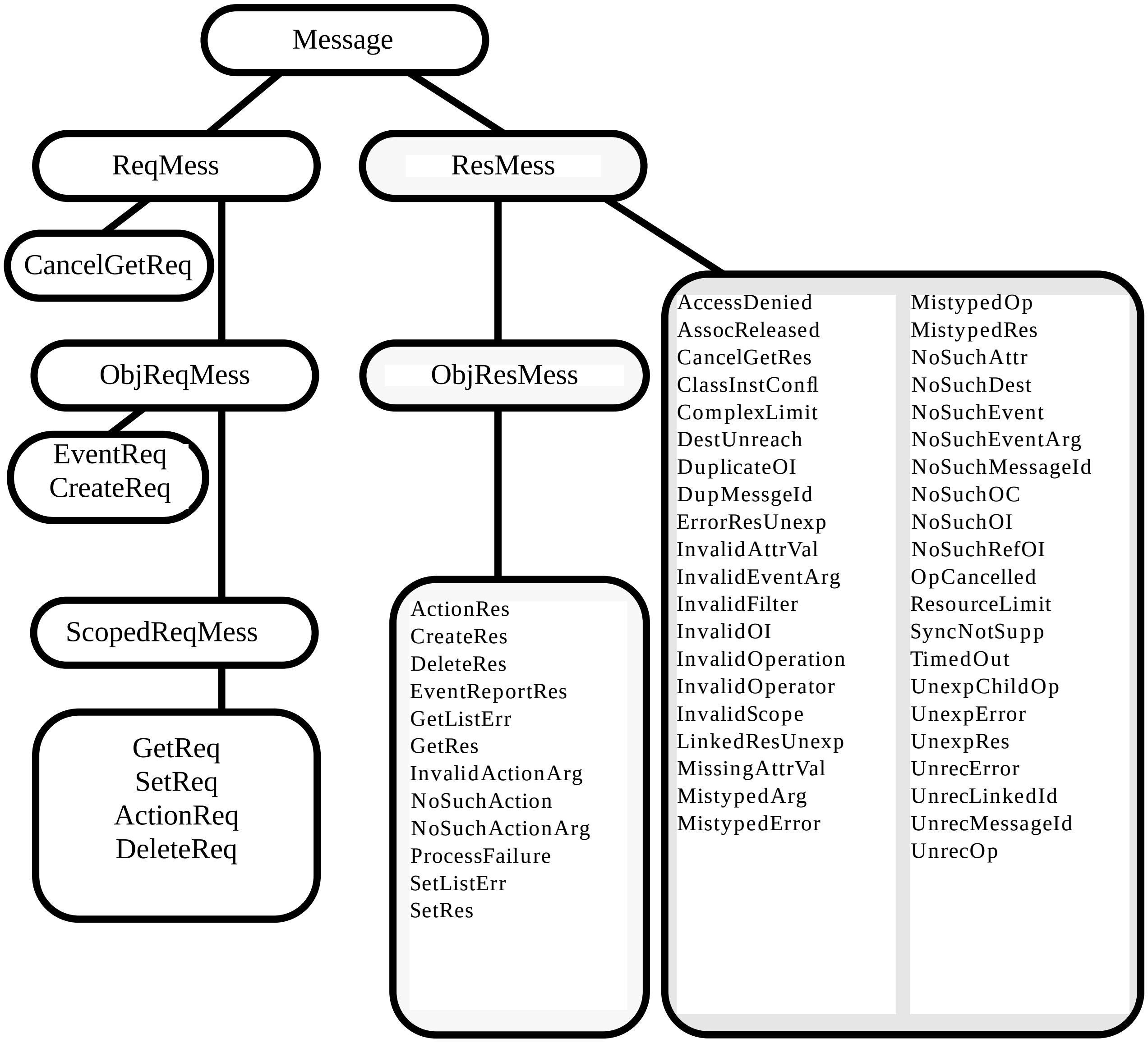


Figure 4-2 Inheritance Tree of the Message Class

Table 4-22 lists the `Message` class public data members:

Table 4-22 Message Class Public Data Members

Type	**Variables**	**Description**
MessId	`id;`	The Message ID, unique to the originator of the message.
Address	`source;`	The address of the Solstice EM module or application that originated this message.
Address	`dest;`	The address of the Solstice EM module or application that is the intended destination of this message. This field is optional when the message is first issued. If this is not supplied, the MRM determines the destination for this message and fills in this field.
MessQOS	`qos;`	The quality of service to be used in processing this request.

Table 4-23 lists the `Message` class method types:

Table 4-23 Message Class Method Types

Functions	**Description**

`type`
`base type`
	Return the message’s type or base type

`is_request`
`is_response`
`is_error`
	Test type of message

`dup`	Make a copy of message
`new_message`	Create new message types
`delete_message`	Delete a message

4.3.29.1 *Constructor*

```
Message(MessType <type>)
```

This constructor, above, takes a `MessType` as its argument and records it as the value of the private variable <*type*>. For the possible values of `MessType`, refer to Section 4.4.7, “MessType,” on page 4-72.

4.3.29.2 *Member Functions of* `Message`

This section describes the member functions of the `Message` class.

basetype

```
MessType basetype()
```

This function call returns the message’s base type.

delete_message

```
static void delete_message(MessagePtr <mp>)
```

This function call deletes the `Message` instance to which <*mp*> points. Use this function as a destructor for messages.

dup

```
virtual MessagePtr dup() = 0
```

This function call creates a duplicate of the message and returns a pointer to the duplicate. Note that the embedded Asn1Values are copied by reference.

is_error

```
vBoolean is_error()
{ return ( <t> > CANCEL_GET_REQ);
}
```

This function call returns true if this is an error message and false if it is not. The message is determined to be an error based on its <*t*> member variable.

is_request

```
Boolean is_request()
{ return ( <t> > CANCEL_GET_REQ);
}
```

This function call returns true if this is a request message and false if it is not. The message is determined to be a request based on its <*t*> member variable.

is_response

```
Boolean is_response()
{ return ( <t> > CANCEL_GET_REQ);
}
```

This function call returns true if this is a response message and false if it is not. The message is determined to be a response based on its <*t*> member variable.

new_message

```
static Message* new_message(MessType <type>)
```

This function call creates a new `Message` type.

type

```
MessType type()
```

This function call returns the type of this message. For the possible values of the returned enumeration, refer to Section 4.4.7, “MessType,” on page 4-72.

4.3.30 MessageSAP Class

Inheritance: `class MessageSAP`

`#include <pmi/message.hh>`

A key class for the low-level use of the PMI is the `MessageSAP` class. The `KernelMessageSAP`, is a subclass of the `MessageSAP` class. The following classes, are also subclasses of `MessageSAP`:

Table 4-24 MessageSAP Subclasses

Subclass Names	**Description**
`ApplMessageSAP`	A transport independent SAP used by applications
`TDApplMessageSAP`	A transport dependent SAP used by applications
`TIMessageSAP`	A transport independent SAP used by the MIS
`TDMessageSAP`	A transport dependent SAP used by the MIS

The `MessageSAP` object class is used as the endpoint of a communications link between two Solstice EM modules. A `MessageSAP` is created on each end of a communications path by a routine called to register a module or application with the MRM. This routine creates both `MessageSaps` and then notifies both parties of the `MessageSAP` that they are to use for communication to the other end. The `MessageSAP` maintains a message id so that request messages issued from this `MessageSAP` can be uniquely identified. In addition, it contains two instances of the `Event` object class. These two instances are the `receive_request` event and the `detach` event. These are used to notify the owner of the `MessageSAP` that a request message has been received and also to notify the owner of the `MessageSAP` that the other side (the `MessageSAP` to which this one is attached) has been deleted.

The `MessageSAP` object class defines a number of member functions which are used to either send or receive messages. In addition, member functions are provided that return a unique message id and cancel a callback for a message or an event.

The `MessageSAP` class defines queues of pointers to messages. The messages pointed to are all subclasses of the `Message` class defined in the `/opt/SUNWconn/em/include/pmi/message.hh` file.

The following figure shows the inheritance hierarchy for the classes based on `MessageSAP`.

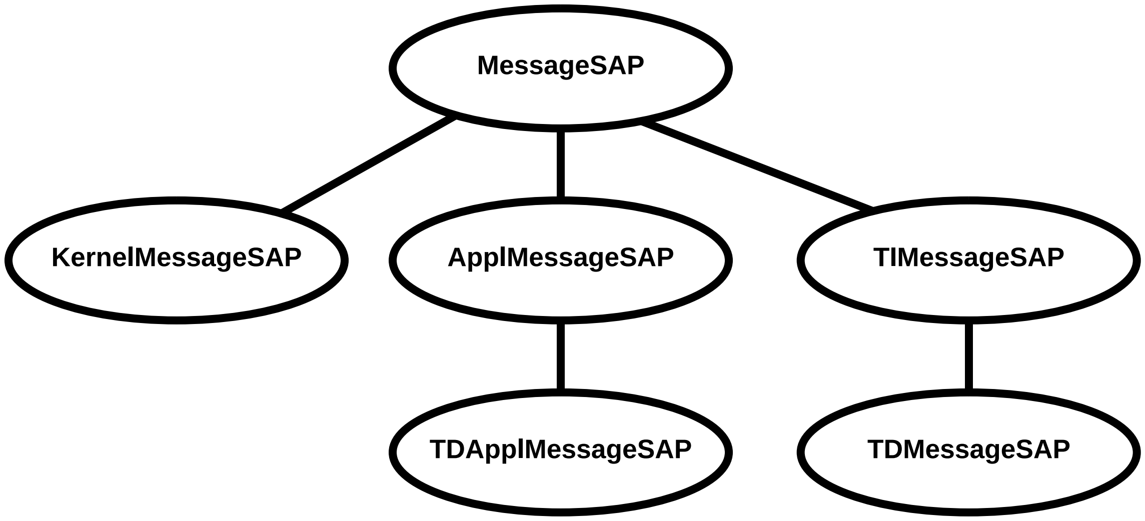


Figure 4-3 Inheritance Tree of the MessageSAP Class

Table 4-25 lists the `MessageSAP` public data members:

Table 4-25 MessageSAP Public Data Members

Type	**Variables**	**Description**
Callback	`receive_request_cb;`	The event to be posted when an incoming request arrives.
Callback	`detach_cb;`	The event to be posted when the message sap detaches.

Table 4-26 lists the `MessageSAP` method types:

Table 4-26 MessageSAP Method Types

Method Name	**Method Type**
`send`	Send a message (with or without blocking).
`receive_request` `receive_response`	Respond to a received request.
`cancel_callback`	Cancel the callback for a pending response.
`new_id`	Generate an ID for a message.

4.3.30.1 *Constructor*

```
MessageSAP ()
```

This example is the constructor for the `MessageSAP`.

4.3.30.2 *Member Functions of* `MessageSAP`

This section describes the member functions of the `MessageSAP` class.

cancel_callback

```
virtual Result cancel_callback(MessId <m_id>) = 0;
```

This function call cancels the pending response callback for a particular message id. This form of the `cancel_callback` member function cancels the callback that was attached to a request sent via the non-blocking form of the `send` member function. This function takes a message id <*m_id*> as input and searches for any callback routine which might have been specified for the response identified by <*m_id*>. The callback is removed from the list of callbacks so that when the response message arrives, it is dropped. It returns `OK` if the callback is was successfully cancelled.

```
virtual Result cancel_callback(Event &<e>) = 0;
```

This function call cancels the pending response callback for any matching message. This form of the `cancel_callback` member function cancels the callback that was attached to a request sent via the non-blocking form of the `send` member function. This function takes an event reference <*e*> as input and searches references to this event in the callback list. If any are found, they are deleted from the list. Later, when the response message arrives, it is dropped.

get_address

```
Result get_address(Asn1Value & <presaddr>,
                   Address & <addr>) ;
```

Used to create addresses. Definitions can be found in the `/opt/SUNWconn/em/include/libsrc/pmi/address.hh` file.

get_ap_addr

```
Result get_ap_addr(const DataUnit & <apaddr>) ;
```

Used to create addresses. Definitions can be found in the `/opt/SUNWconn/em/include/libsrc/pmi/address.hh` file.

make_up_paddr

```
Result make_up_paddr(P_Addr &<addr>,
                     DataUnit & <psel>,
                     DataUnit& <ssel>,
                     IP_Address *<p_addr>,
                     TCP_Port *<p_port>) ;
```

```
Result make_up_paddr(P_Addr & <addr>,
                     DataUnit & <psel>,
                     DataUnit & <ssel>,
                     DataUnit & <tsel>,
                     U32 <nsel_count>,
                     DataUnit & <nsel>) ;
```

Used to create addresses. Definitions can be found in the `/opt/SUNWconn/em/include/libsrc/pmi/address.hh` file.

new_id

```
MessId new_id()
```

This function call generates a new message identifier (one greater than the last ID this `MessageSAP` supplied), stores it privately in the `MessId` instance, and returns its value.

receive_request

```
virtual Result receive_request(MessagePtr &<mp>) = 0;
```

This function call receives the next pending request message. This function is called after a notification has arrived via the `receive_request_event` mechanism. The `receive_request_event` receives a notification that a request message has been queued up for this `MessageSAP` and then this routine should be called to actually access the message. The function takes a reference to a Message pointer and this pointer is set to point to the message received.

```
virtual Result receive_response(MessId &<m_id>,
    MessagePtr &<mp>) = 0;
virtual Result receive_response(ResponseHandle &<rh>,
    MessagePtr &<mp>) = 0;
```

This function call receives the next response for a given message. The message whose response is sought is identified either by its message ID (<*m_id*>) or by a response handle (<*rh*>). This function is called after the owner of the MessageSAP has been notified that a response message has been queued to this MessageSAP. The notification takes place by having the blocking form of the MessageSAP::send return successful for a confirmed request message. This notification can also occur if the event for the non-blocking form of the send member function is notified. The function sets <*mp*> with a pointer to the response message. It returns OK if the first argument successfully identifies a response and the message pointer is successfully set to the response message.

send

```
virtual SendResult send( MessagePtr &<mp>,
MTime <block_time> = INFINITY) = 0;
```

This function call is the blocking form of the send member function. This version of send takes a pointer to an instance of the Message object class and a <*block_time*> parameter. The message pointer points to the message that is to be sent via this MessageSAP. The <*block_time*> parameter specifies how long the caller of this function is willing to wait for the message to be sent. If the message cannot be sent within the time specified, an error is returned.

Messages might not be sent because of resource limitations or a host of other problems.

The above function call returns an error code that specifies why a message could not be sent. The possible values of `SendResult` are:

```
typedef enum SendResult
       { SENT,
       BAD_MESSAGE,
       WOULD_BLOCK,
       NO_MEM
       };
```

If this is a request message and the function returns `SENT`, the request message has been successfully sent and the response to this request has been queued to this `MessageSAP`. If this was a non-confirmed request, `SENT` indicates only that the request has been sent successfully.

If this is a response message, `SENT` indicates that the response has been sent.

The following function call is the non-blocking form of the `send` member function. This version of `send` takes a pointer to an instance of the `Message` object class, an instance of the `Event` object class, and a <*block_time*> parameter. This version of `send` should only be used to send confirmed request messages. It should not be used to send unconfirmed requests or responses.

The message pointer <*mp*> points to the message that is to be sent via this `MessageSAP`. The callback <*cb*> specifies a procedure that is to be called whenever the response for this request message has been queued to this `MessageSAP`. The <*block_time*> parameter specifies how long the caller of this function is willing to wait for the message to be sent. If the message cannot be sent within the time specified, an error is returned.

```
virtual SendResult send(MessagePtr <mp>,
const Callback&<cb>,
                        MTime <block_time> = INFINITY) = 0;
```

It might not be possible to send a message because of resource limitations or a host of other problems. This routine returns an error code that indicates why a message could not be sent. (Refer to `SendResult`, in the discussion of the preceding version of `send`, above.)

set_ap_addr

```
Result set_ap_addr(const DataUnit & <apaddr>,
                        DataUnit & <old_apaddr>) ;
```

Used to create addresses. Definitions can be found in the `/opt/SUNWconn/em/include/libsrc/pmi/address.hh` file.

4.3.30.3 `MessageSAP` *Initialization*

Following is a sample on how to initialize a `MessageSAP`:

```
result init_kernel_msg_sap(Address <source_module>,
                                   MessageSAP **sap_p_p);
```

The parameter, `sap_p_p`, is initialized as a result of the call to `init_kernel_msg_sap`. It is the `MessageSAP` your driver module should use to send and receive messages from the Solstice EM MRM. After `init_kernel_msg_sap` has been called, the `MessageSAP` parameter should be initialized to point to the callback handlers that are used by the MRM.

The function returns a value whose type is `Result`: that is, a boolean value defined as either `OK` or `NOT_OK`. `Result` is defined in the `/opt/SUNWconn/em/include/pmi/sys_type.hh` file.

4.3.31 `MessQOS` *Class*

Inheritance: `class MessQOS`

`#include <pmi/message.hh>`

Method Types: No public member functions are declared in this class.

The `MessQOS` class represents the Quality of Service indicator included in all messages. This class is currently a null class (i.e. it has no member functions or variables). Its purpose is to store data that affects the type of service to be given to a message. This data might include such things as the allowable lifetime of the message and the type of behavior associated with the transportation of this message. Some examples might be:

- request time-out parameters
- security parameters
- reliability / retry parameters

4.3.32 MessScope *Class*

Inheritance: `class MessScope`

`#include <pmi/message.hh>`

Method types: No method types declared in this class.

The `MessScope` class defines a message's scope (that is, the range of objects to which the message is to be applied). An instance of `MessScope` contains a `MessScopeType` variable and an optional level. The enumeration defines five types of scoping: `ALL_LEVELS`.

Table 4-27 Types of MessScope Scoping

Scope	**Description**
`BASE_OBJECT`	A request message containing a scope parameter equal to `BASE_OBJECT` should be sent only to the object specified in the request.
`NTH_LEVEL`	The request message should be sent to those objects which exist N levels below the base object in the Management Information Tree (MIT). The base object is not sent the message, only those objects which are Nth level descendents of the base object. The <*level*> variable in the `MessScope` class is set to the level desired.
`BASE_TO_NTH_LEVEL`	The request message is sent to the base object and all descendents of the base object down to the Nth level. Again, the <*level*> variable is used to indicate the final level of objects that the request is to be routed to.
`ALL_LEVELS`	The request message to be sent to the base object and all of its descendents in the MIT. The <*level*> variable in the `MessScope` class is only used for `NTH_LEVEL` and `BASE_TO_NTH_LEVEL` scoping.
`ALL_LEVELS_EXCEPT_BASE`	Used internally by the Solstice EM MIS.

Table 4-28 lists the `MessScope` public data members:

Table 4-28 MessScope Public Data Members

Type	**Variables**	**Description**
MessScopeType	type;	{`BASE_OBJECT,` `NTH_LEVEL,` `BASE_TO_NTH_LEVEL,` `ALL_LEVELS` }
U32	level;	

4.3.32.1 Constructors

This section specifies the constructors of the `MessScope` class.

`MessScope()`

`MessScope(MessScopeType` *<type>*`, U32` *<level>*`)`

4.3.33 `MissingAttrVal` *Class*

Inheritance: `class MissingAttrVal : public ResMess, public Message, public QueueElem`

`#include <pmi/message.hh>`

Method Types: No public member functions are declared in this class.

In addition to the functions or variables it inherits, the `MissingAttrVal` class adds an attribute ID list parameter. The usage of this parameter is described in detail in the CMIS documentation covering the `missingAttributeValue` error.

Table 4-29 lists the `MissingAttrVal` public variable:

Table 4-29 MissingAttrVal Public Variable

Type	**Variable**	**Description**
Asn1Value	`attr_id_list;`	The list of attribute IDs specified in the Create request. If an attribute is required to be in an object when the object is created and that attribute is not present in this list, then this error message is generated.

4.3.33.1 Constructor

```
MissingAttrVal()
```

This constructor takes no parameters. It only initializes its parent class(es).

4.3.34 `MistypedArg` *Class*

Inheritance: `class MistypedArg: public ResMess, public Message, public QueueElem`

`#include <pmi/message.hh>`

Data Members: No public data members declared in this class.

Method Types: No public member functions are declared in this class.

The `MistypedArg` object class contains all of the member variables and member functions that are present in the classes that it has derived from, either directly or indirectly. This class defines no functions or variables beyond those it inherits. The error message that this class represents is patterned after the ROSE error message mistyped-argument. The usage of this message is described in detail in the documentation covering the ROSE protocol. It is used within the Solstice EM MIS to indicate that one of the arguments supplied with a request message was not supposed to be present.

4.3.34.1 *Constructor*

```
MistypedArg()
```

This constructor takes no parameters. It only initializes its parent class(es).

4.3.35 `MistypedError` *Class*

Inheritance: `class MistypedError : public ResMess, public Message, public QueueElem`

`#include <pmi/message.hh>`

Data Members: No public data members declared in this class.

Method Types: No public member functions are declared in this class.

The `MistypedError` object class contains all of the member data members and member functions that are present in the classes that it has derived from, either directly or indirectly. This class defines no functions or variables beyond those it inherits. The error message that this class represents is patterned after the ROSE error message mistyped-parameter. It is used within the Solstice EM MIS to indicate that an error message, generated in response to a particular request, contained a parameter which either was not expected as part of the error message or which was not formed properly.

4.3.35.1 *Constructor*

```
MistypedError()
```

This constructor takes no parameters. It only initializes its parent class(es).

4.3.36 `MistypedOp` *Class*

Inheritance: `class MistypedOp : public ResMess, public Message, public QueueElem`

`#include <pmi/message.hh>`

Data Members: No public data members declared in this class.

Method Types: No public member functions are declared in this class.

The `MistypedOp` object class contains all of the member variables and member functions that are present in the classes that it has derived from, either directly or indirectly. This class does not add any additional parameters. The usage of this message is described in detail in the CMIS documentation covering the `mistypedOperation` error.

4.3.36.1 *Constructor*

```
MistypedOp()
```

This constructor takes no parameters. It only initializes its parent class(es).

4.3.37 `MistypedRes` *Class*

Inheritance: `class MistypedRes : public ResMess, public Message, public QueueElem`

`#include <pmi/message.hh>`

Data Members: No public data members declared in this class.

Method Types: No public member functions are declared in this class.

The `MistypedRes` object class contains all of the member variables and member functions that are present in the classes that it has derived from, either directly or indirectly. This class defines no functions or variables beyond those it inherits. The error message that this class represents is patterned after the ROSE error message mistyped-result. It is used within the Solstice EM MIS to indicate that a result message, generated for a particular request, contained a parameter that was either not expected or was not formed properly.

4.3.37.1 *Constructor*

```
MistypedRes()
```

This constructor takes no parameters. It only initializes its parent class(es).

4.3.38 `NoSuchAction` *Class*

Inheritance: `class NoSuchAction : public ObjResMess, public Message, public QueueElem`

`#include <pmi/message.hh>`

Method Types: No public member functions are declared in this class.

In addition to the functions or variables it inherits, the `NoSuchAction` class adds two Asn1Value parameters to store a current time and an action type. The usage of these parameters is described in detail in the CMIS documentation covering the `noSuchAction` error.

The <*action_type*> and <*curr_time*> members are only defined when returning an error from a scoped `ACTION_REQ`.

Table 4-30 lists the `NoSuchAction` public data members:

Table 4-30 NoSuchAction Public Data Members

Type	**Variables**	**Description**
Asn1Value	`action_type;`	The invalid action type as extracted from the request message.
Asn1Value	`curr_time`;	An optional parameter specifying the time that this response message was generated.

4.3.38.1 *Constructor*

```
NoSuchAction()
```

This constructor takes no parameters. It only initializes its parent class(es).

4.3.39 `NoSuchActionArg` *Class*

Inheritance: `class NoSuchActionArg : public ResMess, public Message, public QueueElem`

`#include <pmi/message.hh>`

Method Types: No public member functions are declared in this class.

In addition to the functions or variables it inherits, the `NoSuchActionArg` class adds two Asn1Value parameters to store a current time and an action type. The usage of these parameters is described in detail in the CMIS documentation covering the `noSuchArgument` error. There are two choices defined for the `noSuchArgument` error message and this class defines the `actionId` choice (first of the two).

The <*action_type*> and <*curr_time*> members are only defined when returning an error from a scoped `ACTION_REQ`.

Table 4-31 lists the `NoSuchActionArg` public data members:

Table 4-31 NoSuchActionArg Public Data Members

Type	**Variables**	**Description**
Asn1Value	`action_type;`	The invalid action type as extracted from the request message.
Asn1Value	`curr_time`;	An optional parameter specifying the time that this response message was generated.

4.3.39.1 *Constructor*

```
NoSuchActionArg()
```

This constructor takes no parameters. It only initializes its parent class(es).

4.3.40 `NoSuchAttr` *Class*

Inheritance: `class NoSuchAttr : public ResMess, public Message, public QueueElem`

```
#include <pmi/message.hh>
```

Method Types: No public member functions are declared in this class.

In addition to the functions or variables it inherits, the `NoSuchAttr` class adds an attribute ID parameter. The usage of this parameter is described in detail in the CMIS documentation covering the `noSuchAttribute` error.

Table 4-32 lists the `NoSuchAttr` public variable:

Table 4-32 NoSuchAttr Public Variable

Type	**Variable**	**Description**
Asn1Value	`attr_id;`	This is the invalid attribute ID that caused the generation of this error message.

4.3.40.1 Constructor

```
NoSuchAttr()
```

This constructor takes no parameters. It only initializes its parent class(es).

4.3.41 `NoSuchEvent` *Class*

Inheritance: `class NoSuchEvent : public ResMess, public Message, public QueueElem`

```
#include <pmi/message.hh>
```

Method Types: No public member functions are declared in this class.

In addition to the functions or variables it inherits, the `NoSuchEvent` class adds two Asn1Value parameters to store an object class and an event type. The usage of these parameters is described in detail in the CMIS documentation covering the `noSuchEventType` error.

Table 4-33 lists the `NoSuchEvent` public data members:

Table 4-33 NoSuchEvent Public Data Members

Type	**Variables**	**Description**
Asn1Value	`oc;`	The object class for which the event report request was generated.
Asn1Value	`event_type;`	The invalid event type that caused the generation of this error message.

4.3.41.1 Constructor

```
NoSuchEvent()
```

This constructor takes no parameters. It only initializes its parent class(es).

4.3.42 `NoSuchEventArg` *Class*

Inheritance: `class NoSuchEventArg : public ResMess, public Message, public QueueElem`

`#include <pmi/message.hh>`

Method Types: No public member functions are declared in this class.

In addition to the functions or variables it inherits, the `NoSuchEventArg` class adds two Asn1Value parameters to store an object class and an event type. The usage of these parameters is described in detail in the CMIS documentation covering the `noSuchArgument` error. There are two choices defined for the `noSuchArgument` error message and this class defines the `eventId` choice (second of the two).

Table 4-34 lists the `NoSuchEventArg` public data members:

Table 4-34 NoSuchEventArg Public Data Members

Type	**Variables**	**Description**
Asn1Value	`oc;`	The object class for which the event report request was generated.
Asn1Value	`event_type;`	The invalid event type that caused the generation of this error message.

4.3.42.1 Constructor

```
NoSuchEventArg())
```

This constructor takes no parameters. It only initializes its parent class(es).

4.3.43 `NoSuchMessageId` *Class*

Inheritance: `class NoSuchMessageId : public ResMess, public Message, public QueueElem`

`#include <pmi/message.hh>`

Method Types: No public member functions are declared in this class.

In addition to the functions or variables it inherits, the `NoSuchMessageId` class adds a get ID parameter. The usage of this parameter is described in detail in the CMIS documentation covering the `noSuchInvokeId` error.

Table 4-35 lists the `NoSuchMessageId` public variable:

Table 4-35 NoSuchMessageId Public Variable

Type	**Variable**	**Description**
Asn1Value	`get_id;`	This is the invalid message id that caused generation of this error message.

4.3.43.1 Constructor

```
NoSuchMessageId()
```

This constructor takes no parameters. It only initializes its parent class(es).

4.3.44 NoSuchOC Class

Inheritance: `class NoSuchOC : public ResMess, public Message, public QueueElem`

`#include <pmi/message.hh>`

Method Types: No public member functions are declared in this class.

In addition to the functions or variables it inherits, the `NoSuchOC` class adds an Asn1Value parameter to store an object class. The usage of this parameter is described in detail in the CMIS documentation covering the `noSuchObjectClass` error.

Table 4-36 lists the `NoSuchOC` public variable:

Table 4-36 NoSuchOC Public Variable

Type	**Variable**	**Description**
Asn1Value	`oc;`	The invalid object class that caused the generation of this error message.

4.3.44.1 Constructor

```
NoSuchOC()
```

This constructor takes no parameters. It only initializes its parent class(es).

4.3.45 NoSuchOI Class

Inheritance: `class NoSuchOI : public ResMess, public Message, public QueueElem`

`#include <pmi/message.hh>`

Method Types: No public member functions are declared in this class.

In addition to the functions or variables it inherits, the `NoSuchOI` class adds an Asn1Value parameter to store an object instance. The usage of this parameter is described in detail in the CMIS documentation covering the `noSuchObjectInstance` error.

Table 4-37 lists the `NoSuchOI` public variable:

Table 4-37 NoSuchOI Public Variable

Type	**Variable**	**Description**
Asn1Value	`oi;`	The invalid object instance that caused the generation of this error message.

4.3.45.1 Constructor

```
NoSuchOI()
```

This constructor takes no parameters. It only initializes its parent class(es).

4.3.46 `NoSuchRefOI` *Class*

Inheritance: `class NoSuchRefOI : public ResMess, public Message, public QueueElem`

`#include <pmi/message.hh>`

Method Types: No public member functions are declared in this class.

In addition to the functions or variables it inherits, the `NoSuchRefOI` class adds an object instance parameter. The usage of this parameter is described in detail in the CMIS documentation covering the `noSuchReferenceObject` error.

Table 4-38 lists the `NoSuchRefOI` public variable:

Table 4-38 NoSuchRefOI Public Variable

Type	**Variable**	**Description**
Asn1Value	`oi;`	The invalid object instance that caused the generation of this error message.

4.3.46.1 Constructor

```
NoSuchRefOI()
```

This constructor takes no parameters. It only initializes its parent class(es).

4.3.47 `ObjReqMess` *Class*

Inheritance: `class ObjReqMess : public ReqMess, public Message, public QueueElem`

`#include <pmi/message.hh>`

Method Types: No public member functions are declared in this class.

In addition to the functions or variables it inherits, the `ObjReqMess` class adds two Asn1Value parameters for an object class and an object instance. The usage of these fields is defined in the CMIS definition for the type of message indicated.

Table 4-39 lists the `ObjReqMess` public data members:

Table 4-39 ObjReqMess Public Data Members

Type	**Variables**	**Description**
Asn1Value	`oc;`	This is either a base or managed object class as defined for the type of message being created.
Asn1Value	`oi;`	This is either a base or managed object instance as defined for the type of message being created.

4.3.47.1 *Constructor*

```
ObjReqMess(MessType <type>)
```

4.3.48 `ObjResMess` *Class*

Inheritance: `class ObjResMess : public ResMess, public Message, public QueueElem`

`#include <pmi/message.hh>`

Method Types: No public member functions are declared in this class.

In addition to the functions or variables it inherits, the `ObjResMess` class adds two Asn1Values to store an object class and an object instance. Any response (also linked request, error, and linked error) messages which include object class and object instance parameters in the message would be derived from this class. Table 4-40 lists the `ObjResMess` public data members:

Table 4-40 ObjResMess Public Data Members

Type	**Variables**	**Description**
Asn1Value	`oc;`	The object class for this response message. The usage of this variable depends on the type of message which is inherited from this class and is defined in the CMIS documentation for that message type.
Asn1Value	`oi;`	The object instance for this response message. The usage of this variable depends on the type of message which is inherited from this class and is defined in the CMIS documentation for that message type.

4.3.48.1 Constructor

```
ObjResMess (MessType <type>)
```

This constructor takes a `MessType` variable as input and passes this variable on to the constructor(s) for the classes from which this class is derived.

4.3.49 `OpCancelled` Class

Inheritance: `class OpCancelled : public ResMess, public Message, public QueueElem`

`#include <pmi/message.hh>`

Data Members: No public data members declared in this class.

Method Types: No public member functions are declared in this class.

The `OpCancelled` object class contains all of the member variables and member functions that are present in the classes that it has derived from, either directly or indirectly. This class defines no functions or variables beyond those it inherits. The usage of this message is described in detail in the CMIS documentation covering the `operationCancelled` error.

4.3.49.1 Constructor

```
OpCancelled()
```

This constructor takes no parameters. It only initializes its parent class(es).

4.3.50 `ProcessFailure` Class

Inheritance: `class ProcessFailure : public ObjResMess, public Message, public QueueElem`

`#include <pmi/message.hh>`

Method Types: No public member functions are declared in this class.

In addition to the functions or variables it inherits, the `ProcessFailure` class adds a specific error information parameter. The usage of this parameter is described in detail in the CMIS documentation covering the `processingFailure` error.

Table 4-41 lists the `ProcessFailure` public variable:

Table 4-41 ProcessFailure Public Variable

Type	**Function**	**Description**
Asn1Value	`spec_err_info;`	Error information which gives additional information about why this error message was generated. The format of this parameter is variable and depends upon the object class specified in this error message.

4.3.50.1 Constructor

```
ProcessFailure()
```

This constructor takes no parameters. It only initializes its parent class(es).

4.3.51 `ReqMess` *Class*

Inheritance: `class ReqMess : public Message, public QueueElem`

`#include <pmi/message.hh>`

Method Types: No public member functions are declared in this class.

In addition to the functions or variables it inherits, the `ReqMess` class adds a `MessMode` parameter to a message. The usage of `MessMode` variables are described in Section 4.4.2, “MessMode,” on page 4-70.

Table 4-42 lists the `ReqMess` public data members:

Table 4-42 ReqMess Public Data Members

Type	**Variables**	**Description**
MessMode	`mode;`	The mode in which a request message is sent: {CONFIRMED, UNCONFIRMED}
Oid	`app_context`;	The application context name for this request message. This is used to establish an association within the protocol driver.
U32	`flags`;	
enum	`ReqFlags`	{OVERRIDE_NAME_BINDING = 1, OVERRIDE_SET_CHECKS = 2, INTERNAL_RELATIONSHIP_CHANGE = 4 };

4.3.51.1 Constructor

```
protected: ReqMess(MessType <type>)
```

This constructor takes a `MessType` variable as input and sets the member variable <*mode*> equal to the variable passed in.

4.3.52 `ResMess` *Class*

Inheritance: `class ResMess : public Message, public QueueElem`

`#include <pmi/message.hh>`

Method Types: No public member functions are declared in this class.

In addition to the functions or variables it inherits, the `ResMess` class adds a Boolean variable called `linked` that indicates whether this is a linked response message. In all cases, a linked response contains the same data as would be found in an unlinked response so that (applied to the same message) `linked_get_result` returns the same data as `get_result`. Error messages are considered to be response messages, but can also be linked requests.

Table 4-43 lists the ResMess public variable:

Table 4-43 ResMess Public Variable

Type	**Variable**	**Description**
Boolean	`linked;`	When this variable is set to `TRUE`, this is a linked request message. When `FALSE`, this is a response message.

4.3.52.1 Constructor

```
ResMess(MessType <type>)
```

This constructor takes a `MessType` variable as input and passes this variable on to the constructor(s) for the classes from which this class is derived.

4.3.53 `ResourceLimit` *Class*

Inheritance: `class ResourceLimit : public ResMess, public Message, public QueueElem`

`#include <pmi/message.hh>`

Data Members: No public data members declared in this class.

Method Types: No public member functions are declared in this class.

The `ResourceLimit` object class contains all of the member variables and member functions that are present in the classes that it has derived from, either directly or indirectly. This class defines no functions or variables beyond those it inherits. The error message that this class represents is patterned after the ROSE error message resource-limitation. It is used within the Solstice EM MIS to indicate that the receiver of a request message was unable to service the request due to a lack of resources.

4.3.53.1 Constructor

```
ResourceLimit()
```

This constructor takes no parameters. It only initializes its parent class(es).

4.3.54 `ScopedReqMess` Class

Inheritance: `class ScopedReqMess : public ObjReqMess, public ReqMess, public Message, public QueueElem`

`#include <pmi/message.hh>`

Method Types: No public member functions are declared in this class.

In addition to the functions or variables it inherits, the `ScopedReqMess` class adds a `MessScope` variable, two Asn1Value parameters specifying a filter and access control, and a `MessSync` parameter. These parameters are present in all request messages for which scoping can be specified. Each of these parameters is optional and the way in which they are used is defined in the CMIS documentation describing their use.

Table 4-44 lists the `ScopedReqMess` public data members:

Table 4-44 ScopedReqMess Public Data Members

Type	**Variables**	**Description**
MessScope	`scope;`	The type of scoping to be used for this request message. The possible types of scoping are listed on page 4-71.

Table 4-44 ScopedReqMess Public Data Members

Type	**Variables**	**Description**
Asn1Value	`filter`;	This defines a filter that all objects selected via scoping must pass. The message is not sent to any object that does not pass the filter.
Asn1Value	`access`;	This defines the access control that objects selected via scoping must be pass. The message is not sent to any object that does not pass the access control.
MessSync	`sync`;	The type of synchronization for this scoped message; {`ATOMIC`, `BEST_EFFORT` }

4.3.54.1 *Constructor*

```
ScopedReqMess(MessType <type>)
```

This constructor takes a `MessType` variable as input and passes this variable on to the constructor(s) for the classes from which this class derived.

4.3.55 `SetListErr` *Class*

Inheritance: `class SetListErr : public ObjResMess, public Message, public QueueElem`

`#include <pmi/message.hh>`

Method Types: No public member functions are declared in this class.

In addition to the functions or variables it inherits, the `SetListErr` class adds two Asn1Value parameters to store a current time and a set information list. The usage of these parameters is described in detail in the CMIS documentation covering the `SetListError` error. Table 4-45 lists the `SetListErr` public data members:

Table 4-45 SetListErr Public Data Members

Type	Variables	Description
Asn1Value	`set_info_list;`	The list of attributes slated for modification by the Set request. This list contains any attributes which could be modified as well as any attributes which were in error and thus caused the generation of this error message.
Asn1Value	`curr_time`;	An optional parameter specifying the time that this response message was generated.

4.3.55.1 Constructor

```
SetListErr()
```

This constructor takes no parameters. It only initializes its parent class(es).

4.3.56 `SetReq` *Class*

Inheritance: `class SetReq : public ScopedReqMess, public ReqMess, public Message, public QueueElem`

`#include <pmi/message.hh>`

Method Types: No public member functions are declared in this class.

The `SetReq` object class contains all of the member variables and member functions that are present in the classes that it has derived from, either directly or indirectly. In addition to the functions or variables it inherits, the `SetReq` class adds an attribute list parameter. The usage of this parameter is described in detail in the CMIS documentation covering the Set request message.

Table 4-46 lists the `SetReq` public variable:

Table 4-46 SetReq Public Variable

Type	**Variable**	**Description**
Asn1Value	`modify_list;`	Each element of this list contains an attribute ID, an attribute value, and a modify operator.

4.3.56.1 Constructor

```
SetReq()
```

This constructor takes no parameters. It only initializes its parent class(es).

4.3.57 `SetRes` *Class*

Inheritance: `class SetRes : public ObjResMess, public ResMess, public Message, public QueueElem`

`#include <pmi/message.hh>`

Method Types: No public member functions are declared in this class.

In addition to the functions or variables it inherits, the `SetRes` class adds two Asn1Value parameters to store a current time and an attribute list. The usage of these parameters is described in detail in the CMIS documentation covering the Set response. Table 4-47 lists the `SetRes` public data members:

Table 4-47 SetRes Public Data Members

Type	**Variables**	**Description**
Asn1Value	`curr_time;`	An optional parameter specifying the time that this response message was generated.
Asn1Value	`attr_list`	A list of attributes was specified in the Set request message for which this response is being generated. This parameter basically echoes back to the requester the list of attributes that were modified and their new values.

4.3.57.1 *Constructor*

```
SetRes()
```

This constructor takes no parameters. It only initializes its parent class(es).

4.3.58 `SyncNotSupp` *Class*

Inheritance: `class SyncNotSupp : public ResMess, public Message, public QueueElem`

`#include <pmi/message.hh>`

Method Types: No public member functions are declared in this class.

In addition to the functions or variables it inherits, the `SyncNotSupp` class adds a sync parameter. The usage of this parameter is described in detail in the CMIS documentation covering the `syncNotSupported` error.

Each of the classes derived from Message relies on the ISO specifications of the CMIP protocol and ASN.1 data encoding. Table 4-48 lists the `SyncNotSupp` public variable:

Table 4-48 SyncNotSupp Public Variable

Type	**Variable**	**Description**
MessSync	`sync;`	Specifies the type of synchronization which was not able to be performed and caused the generation of this error message.

4.3.58.1 *Constructor*

```
SyncNotSupp()
```

This constructor takes no parameters. It only initializes its parent class(es).

4.3.59 `TimedOut` Class

Inheritance: `class TimedOut : public ResMess, public Message, public QueueElem`

`#include <pmi/message.hh>`

Data Members: No public data members declared in this class.

Method Types: No public member functions are declared in this class.

This class defines no functions or variables beyond those it inherits. The error message that this class represents is generated whenever the life-span of a message has been exceeded. Future functionality envisioned for the Solstice EM MIS would be to include in the `MessQOS` (quality of service) class some indication of how long the requester is willing to wait for a response to a given request message (a message lifetime). A `TimedOut` message would be generated whenever the lifetime for a given request message had been exceeded (i.e. whenever the request has not been responded to within its lifetime).

4.3.59.1 Constructor

```
TimedOut()
```

This constructor takes no parameters. It only initializes its parent class(es).

4.3.60 `UnexpChildOp` Class

Inheritance: `class UnexpChildOp : public ResMess, public Message, public QueueElem`

`#include <pmi/message.hh>`

Data Members: No public data members declared in this class.

Method Types: No public member functions are declared in this class.

The `UnexpChildOp` object class contains all of the member variables and member functions that are present in the classes that it has derived from, either directly or indirectly. This class defines no functions or variables beyond those it inherits. The error message that this class represents is patterned after the ROSE error message unexpected-child-operation. It is used within the Solstice EM MIS to indicate that a linked request was received and that the linked id specified did not refer to a request for which this type of linked reply is valid.

4.3.60.1 Constructor

```
UnexpChildOp()
```

This constructor takes no parameters. It only initializes its parent class(es).

4.3.61 `UnexpError` *Class*

Inheritance: `class UnexpError : public ResMess, public Message, public QueueElem`

`#include <pmi/message.hh>`

Data Members: No public data members declared in this class.

Method Types: No public member functions are declared in this class.

The `UnexpError` class defines no functions or variables beyond those it inherits. The error message that this class represents is patterned after the ROSE error message unexpected-error. It is used within the Solstice EM MIS to indicate that an error message, generated in response to a particular request, is not one of the set of error messages that can be sent in response to that request.

4.3.61.1 Constructor

```
UnexpError()
```

This constructor takes no parameters. It only initializes its parent class(es).

4.3.62 `UnexpRes` *Class*

Inheritance: `class UnexpRes : public ResMess, public Message, public QueueElem`

`#include <pmi/message.hh>`

Data Members: No public data members declared in this class.

Method Types: No public member functions are declared in this class.

The `UnexpRes` object class defines no functions or variables beyond those it inherits. The error message that this class represents is patterned after the ROSE error message result-response-unexpected. It is used within the Solstice EM MIS to indicate that a result message was generated for as non-confirmed request.

4.3.62.1 Constructor

```
UnexpRes()
```

This constructor takes no parameters. It only initializes its parent class(es).

4.3.63 `UnrecError` *Class*

Inheritance: `class UnrecError : public ResMess, public Message, public QueueElem`

`#include <pmi/message.hh>`

Data Members: No public data members declared in this class.

Method Types: No public member functions are declared in this class.

The `UnrecError` class defines no functions or variables beyond those it inherits. The error message that this class represents is patterned after the ROSE error message unrecognized-error. It is used within the Solstice EM MIS to indicate that an error message, generated in response to a particular request, is not one of the set of error messages known within Solstice EM.

4.3.63.1 *Constructor*

```
UnrecError()
```

This constructor takes no parameters. It only initializes its parent class(es).

4.3.64 `UnrecLinkedId` *Class*

Inheritance: `class UnrecLinkedId : public ResMess, public Message, public QueueElem`

`#include <pmi/message.hh>`

Data Members: No public data members declared in this class.

Method Types: No public member functions are declared in this class.

The `UnrecLinkedId` class defines no functions or variables beyond those it inherits. The error message that this class represents is patterned after the ROSE error message "unrecognized-linked-id." It is used within the Solstice EM MIS to indicate that a linked request could not be serviced because the linked id specified in the request did not refer to any known outstanding request.

4.3.64.1 *Constructor*

```
UnrecLinkedId()
```

This constructor takes no parameters. It only initializes its parent class(es).

4.3.65 `UnrecMessageId` *Class*

Inheritance: `class UnrecMessageId : public ResMess, public Message, public QueueElem`

`#include <pmi/message.hh>`

Data Members: No public data members declared in this class.

Method Types: No public member functions are declared in this class.

The `UnrecMessageId` class defines no functions or variables beyond those it inherits. The error message that this class represents is patterned after the ROSE error message unrecognized-invocation. It is used within the Solstice EM MIS to indicate that a result or error message was generated where the message id specified did not refer to any outstanding request.

4.3.65.1 Constructor

```
UnrecMessageId()
```

This constructor takes no parameters. It only initializes its parent class(es).

4.3.66 `UnrecOp` *Class*

Inheritance: `class UnrecOp : public ResMess, public Message, public QueueElem`

`#include <pmi/message.hh>`

Data Members: No public data members declared in this class.

Method Types: No public member functions are declared in this class.

The `UnrecOp` object class defines no functions or variables beyond those it inherits. The error message that this class represents is patterned after the ROSE error message “unrecognized operation.” The usage of this message is described in detail in the documentation covering the ROSE protocol. It is used within the Solstice EM MIS to indicate that the operation requested is not known to the receiver of the request.

Constructor

```
UnrecOp()
```

This constructor takes no parameters. It only initializes its parent class(es).

4.4 Constants and Defined Types

The following subsections describe the constants and defined types for the low-level usage of the PMI.

- *MessId*
- *MessMode*
- *MessagePtr*
- *MessScopeType*
- *MessSync*
- *MessBaseType*
- *MessType*
- *MESSTYPE_MAX*
- *ResponseHandle*
- *SendResult*

4.4.1 MessId

A `MessId` variable is included in each message passed by way of a `MessageSAP`. The `MessId` variable is used to uniquely identify outstanding request messages for a Solstice EM module or application. Each Solstice EM module is responsible for generating new request messages that have unique message IDs. Unique means that the message can not have the same ID as another request that is still outstanding from this module or application. The `MessId` variable is declared in the `/opt/SUNWconn/em/include/pmi/message.hh` file:

```
typedef I32 MessId;
```

4.4.2 MessMode

The `MessMode` variable is declared in the `/opt/SUNWconn/em/include/pmi/message.hh` file:

```
enum MessMode
   {CONFIRMED,
    UNCONFIRMED };
```

4.4.3 MessagePtr

The `MessagePtr` variable is declared in the `/opt/SUNWconn/em/include/pmi/message.hh` files:

```
typedef class Message *MessagePtr;
```

4.4.4 MessScopeType

The `MessScopeType` variable is declared in the `/opt/SUNWconn/em/include/pmi/message.hh` file:

```
enum MessScopeType
   {BASE_OBJECT,
    NTH_LEVEL,
    BASE_TO_NTH_LEVEL,
    ALL_LEVELS,
    ALL_LEVELS_EXCEPT_BASE };
```

4.4.5 MessSync

This enumeration defines the types of message synchronization that can be requested: `ATOMIC` or `BEST_EFFORT`. Currently only `BEST_EFFORT` is supported, since the requirements for `ATOMIC` synchronization are not fully defined by the standards organizations.

Type of Synchronization	**Description**
`BEST_EFFORT`	Scoped requests are to be processed in a best-effort fashion; if one part of a scoped request fails, the other parts of the scoped request are still attempted.
`ATOMIC`	If it were operational, would indicate that if any portion of a scoped request failed, subsequent parts of the request should not be attempted and already completed parts should be reversed.

The `MessSync` variable is declared in the `/opt/SUNWconn/em/include/pmi/message.hh` file:

```
enum MessSync
   {BEST_EFFORT,
    ATOMIC };
```

4.4.6 MessBaseType

The `MessBaseType` variable is declared in the `/opt/SUNWconn/em/include/pmi/message.hh` file:

```
enum MessBaseType
   {MESSAGE,
    REQ_MESS,
    OBJ_REQ_MESS,
    SCOPED_REQ_MESS,
    RES_MESS,
    OBJ_RES_MESS };
```

4.4.7 MessType

The `MessType` enumeration defines a unique ID for each type of message that can be sent within Solstice EM. This includes all the CMIS request messages, CMIS response messages, CMIS error messages, ROSE user-reject responses, SNMP requests, SNMP responses, and Solstice EM error responses. Following is an exhaustive list of the messages it is possible to send via the `MessageSAP` interface. Each of these messages derives from one of the base message classes defined in `/opt/SUNWconn/em/include/pmi/message.hh`:

```
enum MessType
   {

// requests
   EVENT_REPORT_REQ
   GET_REQ
   SET_REQ
   ACTION_REQ
   CREATE_REQ
```

```
    DELETE_REQ
    CANCEL_GET_REQ

// normal responses
    EVENT_REPORT_RES
    GET_RES
    SET_RES
    ACTION_RES
    CREATE_RES
    DELETE_RES
    CANCEL_GET_RES

// Errors
    NO_SUCH_OC
    NO_SUCH_OI
    ACCESS_DENIED
    SYNC_NOT_SUPP
    INVALID_FILTER
    NO_SUCH_ATTR
    INVALID_ATTR_VAL
    GET_LIST_ERR
    SET_LIST_ERR
    NO_SUCH_ACTION
    PROCESS_FAILURE
    DUPLICATE_OI
    NO_SUCH_REF_OI
    NO_SUCH_EVENT
    NO_SUCH_ACTION_REQ
    NO_SUCH_EVENT_ARG
    INVALID_ACTION_ARG
    INVALID_SCOPE
    INVALID_OI
    MISSING_ATTR_VAL
    CLASS_INST_CONFL
    COMPLEX_LIMIT
    MISTYPED_OP
    INVALID_OPERATION
    INVALID_OPERATOR
    NO_SUCH_MESSAGE_ID
    OP_CANCELLED

// ROSE level user-reject responses
    DUP_MESSAGE_ID
    UNREC_OP
```

```
   MISTYPED_ARG
   RESOURCE_LIMIT
   ASSOC_RELEASED
   UNREC_LINKED_ID
   LINKED_RES_UNEXP
   UNEXP_CHILD_OP
   UNREC_MESSAGE_ID
   UNEXP_RES
   MISTYPED_RES
   ERROR_RES_UNEXP
   UNREC_ERROR
   UNEXP_ERROR
   MISTYPED_ERROR

// Solstice EM error responses
   TIMED_OUT
   DEST_UNREACH
   NO_SUCH_DEST
   };
```

4.4.8 MESSTYPE_MAX

The MESSTYPE_MAX variable is declared in the /opt/SUNWconn/em/include/pmi/message.hh file:

```
const MessType MESSTYPE_MAX = NO_SUCH_DEST;
```

4.4.9 ResponseHandle

The ResponseHandle variable is declared in the /opt/SUNWconn/em/include/pmi/message.hh file:

```
typedef void *ResponseHandle;
```

4.4.10 SendResult

The `SendResult` variable is declared in the `/opt/SUNWconn/em/include/pmi/message.hh` file:

```
typedef enum SendResult
      {SENT,
       BAD_MESSAGE,
       WOULD_BLOCK,
       NO_MEM };

// Used by the MessageSAP class.

#define SENT TRUE
```

Nerve Center Interface Library 5

5.1 Introduction

The Nerve Center Interface (NCI) library is built on top of the Portable Management Interface (PMI). The NCI library allows applications to create template requests, launch the request against Management Information Server (MIS) objects, and retrieve information about objects.

A template is a type of management request; it is used to manage a set of selected objects in the MIS. A template is comprised of state, condition, transition, and action requests. Template definitions reside in the MIS.

When a request is launched, the Nerve Center (NC) polls remote objects and retrieves the requested information. If the application transitions to a new state, the managed object's severity and state attributes are changed in the MIS and appropriate applications are informed of that object's state change.

Note – When compiling NCI clients, do not disable C++ exceptions (`-noex` option with Sun C++ compilers) and do link NCI clients with the RogueWave library.

5.2 Class and Function Summary

The following table summarized the classes and functions in this chapter:

Table 5-1 Nerve Center Classes and Functions

Class/Function	**Type**	**Description**
NCAsyncResIterator Class	Class	Allows iteration through the responses received for an asynchronous launch
NCParsedReqHandle Class	Class	Parses request handles returned by NCI functions
NCTopoInfoList Class	Class	Builds a list of toponode information
nci_action_add Function	Function - Result	Adds the action to the transition
nci_action_delete Function	Function - Result	Deletes the action from the list of actions for transition
nci_async_request_start Function	Function - Waiter	Launches NC requests asynchronously
nci_condition_add Function	Function - Result	Creates a condition object that can be used for building templates
nci_condition_delete Function	Function - Result	Deletes an existing condition
nci_condition_get Function	Function - char*	Returns an ASCII string containing RCL statement(s)
nci_error_reason Function	Function - extern	Queries a global variable for error diagnostic data
nci_init Function	Function - Result	Initializes the NCI library
nci_parse_handle Function	Function - Result	Parses a given request handle and returns data
nci_pollrate_add Function	Function - Result	Creates a new pollrate object in the MIS
nci_pollrate_delete Function	Function - Result	Deletes an existing pollrate object in the MIS
nci_request_delete Function	Function - Result	Deletes a running request
nci_request_dump Function	Function - Array(DU)	Returns current state, severity, and variable data
nci_request_info Function	Function - Result	Returns state name, severity string, and severity value
nci_request_list Function	Function - Array(DU)	Returns an array of request handles

Table 5-1 Nerve Center Classes and Functions

Class/Function	**Type**	**Description**
nci_request_start Function	Function - (DU)	Starts a management request against an object
nci_severity_add Function	Function - Result	Creates a new security level in the MIS
nci_severity_delete Function	Function - Result	Deletes an existing severity in the MIS
nci_state_add Function	Function - Result	Adds state, poll rate, severity, and description to a template
nci_state_delete Function	Function - Result	Deletes the state from a template
nci_state_get Function	Function - NC_State	Gets the handle to identify the state in a template
nci_template_add Function	Function - Result	Edits a template
nci_template_copy Function	Function - Result	Copies a template
nci_template_create Function	Function - Result	Creates a handle for a request template
nci_templete_delete Function	Function - Result	Deletes an existing request template from the MIS
nci_templae_find Function	Function - NC_Defn	Gets the handle for an existing request for editing
nci_template_revert Function	Function - NC_Defn	Undos changes done to the template
nci_template_store Function	Function - Result	Stores the template
nci_transition_add Function	Function - Result	Adds a transition to an existing template
nci_transition_delete Function	Function - Result	Deletes a transition between two states
nci_transition_find Function	Function - NC_Transition	Gets the transition in a template
nci_transition_get Function	Function - NC_Transition	Gets the handle on transition from the state
topoNodeId Argument	Argument - U_long	Accepts <topoNodeId> in the form of a DataUnit

5.3 NC Requests

5.3.1 Synchronous Launches

Synchronous launches imply that associated functions do not return until after all the parameter validation and communication with the MIS, regarding the start of the request, is complete. These function do not actually confirm whether the request has started successfully or not. In the sense, it is asynchronous. In comparison, functions in an asynchronous launch return even before any communication with the MIS, regarding the start of the request, takes place.

The functions specifically associated with synchronous launches are:

- *nci_request_delete Function*
- *nci_request_dump Function*
- *nci_request_info Function*
- *nci_request_list Function*
- *nci_request_start Function*

For the many of the request functions use the variable, <*handle*>. <*handle*> is an optional user-defined string. When it is used, the NCI uses it to build the request handle that the NCI returns. The purpose of using <*handle*> is to allow an application, like the auto daemon, to get a list of running requests from the NCI and know which ones it has started. Such a list is useful when an application restarts. An application does not *have* to use <*handle*>; it can keep a list of the requests it has started instead. If <*handle*> is not passed, the request handle is formed by the NCI without it.

5.3.2 Asynchronous Launches

In asynchronous launches, associated functions return before communication with the MIS, regarding the start of the request, takes place. In comparison, synchronous launches imply that associated functions do not return until after all the parameter validation and communication with the MIS, regarding the start of the request, is complete.

The classes/functions associated with asynchronous launches are:

- *NCAsyncResIterator Class*
- *nci_async_request_start Function*

5.4 NCI Library Classes

The library classes included in the NCI are:

Table 5-2 NCI Library Classes

NCI Classes
NCAsyncResIterator Class
NCParsedReqHandle Class
NCTopoInfoList Class

5.4.1 `NCAsyncResIterator` *Class*

The `NCAsyncResIterator` class allows you to iterate through the responses received for an asynchronous launch of possible multiple requests. Each iteration allows you to extract response information about one request.

5.4.1.1 Constructor

```
NCAsyncResIterator::NCAsyncResIterator(
    Ptr <call_data>
);
```

This constructor takes <*call_data*> as an argument. <*call_data*> is the data supplied by nci_async_request_start() when nci_async_request_start() calls the user-installed callback.

5.4.1.2 Destructor

```
NCAsyncResIterator::~NCAsyncResIterator();
```

5.4.1.3 operator++

```
RWBoolean NCAsyncResIterator::operator++();
```

The prefix increment operator advances the iterater one position in the response list. Returns FALSE, if it advances past the end of the response list. Otherwise, it returns TRUE.

5.4.1.4 Member Functions

get_req_handle

```
const DataUnit& NCAsyncResIterator::get_req_handle() const;
```

Returns the request handle for the request determined by the current position of the iterater in the response list. The results are undefined if the iterater is no longer valid or if operator++() has previously returned FALSE.

get_req_status

```
NCAsyncReqStatus NCAsyncResIterator::get_req_status() const;
```

Returns the status of the request determined by the current position of the iterater in the response list. The results are undefined if the iterater is no longer valid or if operator++() has previously returned FALSE.

NCAsyncReqStatus represents the status of the request that is launched asynchronously.

```
enum NCAsyncReqStatus
{
    NOT_INITIALIZED,
    AWAITING_RESPONSE,
    LAUNCH_SUCCESS,
    LAUNCH_FAILURE
};
```

`NOT_INITIALIZED`: Indicates the NC request was never launched.

`AWAITING_RESPONSE`: Indicates that a request (internal CMIP request) has been made to the MIS to launch the NC request, and is waiting for a response from the MIS.

`LAUNCH_SUCCESS`: The NC request was successfully launched.

`LAUNCH_FAILURE`: The NC request launch failed.

get_error_reason

```
const RWCString& NCAsyncResIterator::get_error_reason() const;
```

Returns the reason for failure to launch a given request, determined by the current position of the iterater in the response list. You should use this member function when get_req_status() returns `LAUNCH_FAILURE`.

The results are undefined if the iterater is no longer valid or if operator++() has previously returned `FALSE`.

5.4.2 `NCParsedReqHandle` *Class*

`NCParsedReqHandle` class is responsible for parsing request handles returned by NCI functions. For backward compatibility, nci_parse_handle() is retained. However, it’s recommended that `NCParsedReqHandle` be used wherever possible. This provides better insulation against any changes to the request handle implementation and more information can be extracted from the request handle.

5.4.2.1 Constructor

```
NCParsedReqHandle(const DataUnit& <req_handle_du>);
```

This constructor parses the `<req_handle_du>` which is received byNCI functions that launch requests, either synchronously or asynchronously.

If the request handle that is passed is invalid, the NCParsedReqHandle constructor throws an NCException to the user. To determine the cause of the exception, use `NCException::why()`, which returns `char*`.

5.4.2.2 Default Destructor

```
~NCParsedReqHandle();
```

5.4.2.3 Member Functions

get_topo_id

```
u_long NCParsedReqHandle::get_topo_id() const;
```

This function returns the `topo_id` from the `request_handle`.

get_mis_name

```
const RWCString& NCParsedReqHandle::get_mis_name() const;
```

This function returns the name of the MIS against which the request was launched.

get_template_name

```
const RWCString& NCParsedReqHandle::get_template_name() const;
```

This function returns the name of the template that was used to launch the request.

get_invoke_id

```
u_long NCParsedReqHandle::get_invoke_id() const;
```

This function returns the invokes id of the asynchronous launch. Invoke id is per process and per invocation of nci_async_request_launch(). For synchronous launches, invoke id is always zero.

get_user_stub

```
const RWCString& NCParsedReqHandle::get_user_stub() const;
```

This function returns the <*user_stub*> that was used in the construction of the request handle. This <*user_stub*> is passed to the invocation of synchronous versions of request launches.

5.4.3 `NCTopoInfoList` *Class*

`NCTopoInfoList` class is used to build a list of toponode information. Information about individual toponodes is shared using the copy constructor and assignment operator.

5.4.3.1 Default constructor

```
NCTopoInfoList::NCTopoInfoList();
```

This function is the default constructor that constructs an empty topology information list.

5.4.3.2 Copy constructor

```
NCTopoInfoList::NCTopoInfoList(
    const NCTopoInfoList& <that>
);
```

This is the copy constructor that increments the reference count on the internal implementation object contained within <*that*> object.

5.4.3.3 Destructor

```
NCTopoInfoList::~NCTopoInfoList();
```

This destructor decrements the reference count on the internal implementation object and deletes the internal implementation object when the reference count goes to zero.

5.4.3.4 Operator=

```
NCTopoInfoList& NCTopoInfoList::operator=(
    const NCTopoInfoList& <that>
);
```

The assignment operator increments the reference count on the internal implementation object contained within <*that*> object.

5.4.3.5 Member Functions

add_topo_info

```
Result NCTopoInfoList::add_topo_info(
    u_long <topo_id>,
    Array(DU)& <fdn_set>,
    CDU <mis_name> = null_du
);
```

This member function adds information about one toponode to the NCTopoInfoList object. This overloaded member function is provided for the sake of efficiency and performance. Some applications might already cache the fdns of the toponodes (the value of the topoNodeMOSet GDMO attribute). Such applications can pass in the <*fdn_set*> (optional). Each fdn in the <*fdn_set*> should be either in the slash or brace format. Because `nci_async_request_start()` does not issue any requests to the MIS to get information about the toponodes, performance gains are achieved.

If <*mis_name*> is not specified, it's assumed to be the local MIS (the MIS to which nci_init() initially connected). Returns `OK` if successful. Otherwise, it returns `NOT_OK`.

5.5 NCI Library Functions

Note – All functions that return `Result` are either `OK` or `NOT_OK`. All functions that return Boolean are either `TRUE` or `FALSE`.

Several NCI functions require an argument <mis_name> in this release, as the NCI supports MIS to MIS awareness. NCI accepts <mis_name> in the following formats:

- slash format (/systemid=name:"sol")
- string name ("sol")

<mis_name> is the name of the MIS on which the given NCI function is effective. Unless otherwise specified, <mis_name> automatically maps to the local MIS that the NCI application is connected to.

The NCI library functions included are:

Table 5-3 Nerve Center Library Functions

Function	**Function**
nci_action_add Function	*nci_severity_add Function*
nci_action_delete Function	*nci_severity_delete Function*
nci_async_request_start Function	*nci_state_add Function*
nci_condition_add Function	*nci_state_delete Function*
nci_condition_delete Function	*nci_state_get Function*
nci_condition_get Function	*nci_template_add Function*
nci_error_reason Function	*nci_template_copy Function*
nci_init Function	*nci_template_create Function*
nci_parse_handle Function	*nci_templete_delete Function*
nci_pollrate_add Function	*nci_templae_find Function*
nci_pollrate_delete Function	*nci_template_revert Function*
nci_request_delete Function	*nci_template_store Function*
nci_request_dump Function	*nci_transition_add Function*
nci_request_info Function	*nci_transition_delete Function*
nci_request_list Function	*nci_transition_find Function*
nci_request_start Function	*nci_transition_get Function*

5.5.1 `topoNodeId` *Argument*

Several of the NCI functions include the argument <*topoNodeId*>. NCI accepts <*topoNodeId*> in the form of a DataUnit and supports either of the following formats:

```
u_long topoNodeId;
```

```
DataUnit duTopoNodeId = DateUnit::printf("%u", topoNodeId);
```

The topoNodeId should be constructed as shown in the above syntax before passing to NCI any of the functions that require <*topoNodeId*>. You can use `sscanf()` or `sprintf()` to simulate a comparable effect.

- <*mis_name*>:<*topoId*>
- <*topoId*>

5.5.2 `nci_action_add` *Function*

```
Result
nci_action_add(
    NC_Transition&<transq>,
    const char*<action>,
    const char*<arg0>,
    const char*<arg1>,
    CDU <mis_name>= null_du
);
```

The function `nci_action_add()` adds the action <*action*> to the transition <*transq*>. The action can be either a condition, or mail, or unixcmd. For condition, `arg0` is the name of the condition and `arg1` is null. For mail, `arg0` is the address and `arg1` is the message. For unixcmd, `arg0` is the unix command name and `arg1` is the argument. <*transq*> is the handle on transition as returned by the function `nci_transition_find()`. The function returns `TRUE` if the action is added, `FALSE` if there is an error.

5.5.3 nci_action_delete Function

```
Result
nci_action_delete(
    NC_Transition&<transq>,
    const char*<action>,
    const char*<arg0>,
    const char*<arg1>
);
```

The function `nci_action_delete()` deletes the action <*action*> from the list of actions for transition <*transq*>. The action can be either a condition, or mail, or unixcmd. For condition, <*arg0*> is the name of the condition and <*arg1*> is null. For mail, <*arg0*> is the address and <*arg1*> is the message. For unixcmd, <*arg1*> is the list of arguments specified by <*arg0*>. <*arg1*> is in string format. <*transq*> is the handle on transition as returned by the function `nci_transition_find()`. The function returns TRUE if the action is deleted, FALSE if there is an error.

5.5.4 nci_async_request_start Function

```
Waiter
nci_async_request_start(
    const char*      <template_name>,
    NCTopoInfoList   <topo_info_list>,
    Callback         <user_cb>,
    Timeout          <timeout_time = 3600.0>
);
```

nci_async_request_start() returns a waiter that represents the asynchronous launch.

<*template_name*> is the name of the template that is used to launch the request.

<*topo_info_list*> is information about the toponodes against each of which the request is to be launched asynchronously.

<*user_cb*> is the callback, if installed by the user, that is called when the asynchronous launch completes, either successfully or not. <*user_cb*> is called by NCI. The <*call_data*> parameter of the callback is a pointer that must be used to construct NCAsyncResIterator object. If the <call_data> pointer is not used to construct the NCAsyncResIterator object, a memory leak may occur.

<*timeout_time*> is the timeout interval after which nci_async_request_start() times out and returns. Currently, it's recommended that you use the default timeout. Timeout for lower timer intervals is not supported in this release.

Note – Waiter cancellation or waiter clobbering is not supported in this release.

5.5.5 nci_condition_add *Function*

```
Result
nci_condition_add(
    const char*<condition_name>,
    const char*<condition_desc>,
    const char*<condition>,
    CDU <mis_name> = null_du
);
```

The function nci_condition_add() creates a condition object that can then be used for building request templates. The condition itself is one or more lines separated by new lines specified in the Request Condition Language (RCL). The RCL is documented in the *Solstice Enterprise Manager Reference Manual*. An example of the RCL is in the sample program request_template.cc in Code Example 5-1.

5.5.6 `nci_condition_delete` *Function*

```
Result
nci_condition_delete(
    const char*<condition_name>,
    CDU <mis_name> = null_du
);
```

The function `nci_condition_delete()` deletes an existing condition. If a condition is associated with a transition it cannot be deleted until the transition(s) are deleted.

5.5.7 `nci_condition_get` *Function*

```
char*
nci_condition_get(
    const char*<condition_name>,
    char*<conditionDesc>,
    CDU <mis_name> = null_du
);
```

The function `nci_condition_get()` returns an ASCII string containing a Request Condition Language statement or statements.

5.5.8 `nci_error_reason` *Function*

```
extern DataUnit nci_error_reason;
```

NCI has a facility that allows you to diagnose errors using nci_error_reason. This global variable can be used to know the reason for any failures of any NCI function. This variable contains a meaningful reason only immediately after any NCI function returns an error. This variable is not re-initialized by NCI every time an NCI function is invoked. It is set only when NCI encounters an error.

5.5.9 `nci_init` *Function*

```
Result
nci_init(
    const char*<location>,
    DU& <errorMsg>
);
```

```
Result
nci_init
    Platform&<platform>,
    DU& <errorMsg>
);
```

The function `nci_init()` is an NCI library initialization routine and takes either a hostname (<location>) or a platform object (<platform>). The NCI library connects to the MIS on that host and keeps the Platform object internal. The NCI library assumes the platform object is connected to an MIS and uses the platform instance <`platform`> to talk to the MIS.

5.5.10 `nci_parse_handle` *Function*

```
Result
nci_parse_handle(
    const DataUnit&<request_handle>,
    DataUnit&<template_name>,
    DataUnit&<topoNodeId>
);
```

The function `nci_parse_handle()` parses a given request handle and returns the <*template_name*> and the information about <*topoNodeId*> against which the request was launched. The <*request_handle*> must be a valid one returned by a previous relevant NCI function. It returns `OK` if parsing succeeds. Otherwise, it returns `NOT_OK` and <*template_name*> and <*topoNodeId*> are undefined.

5.5.11 `nci_pollrate_add` *Function*

```
Result
nci_pollrate_add(
    const char*<pollrate_name>,
    int <rate>,
    CDU <mis_name> = null_du
);
```

The function `nci_pollrate_add()` creates a new pollrate object in the MIS. This pollrate can be used in creating requests templates. The pollrate is in seconds.

5.5.12 `nci_pollrate_delete` *Function*

```
Result
nci_pollrate_delete(
    const char*<pollrate_name>,
    CDU <mis_name> = null_du
);
```

The function `nci_pollrate_delete()` deletes an existing pollrate object in the MIS.

5.5.13 `nci_request_delete` *Function*

```
Result
nci_request_delete(
    CDU <request_handle>
);
```

The function `nci_request_stop()` deletes a running request. The handle for the request must be passed. The handle can be obtained from either `nci_request_start()` or by `nci_request_list()`. These request calls are unconfirmed and asynchronous.

5.5.14 `nci_request_dump` *Function*

```
Array(DU)
nci_request_dump(
    CDU <request_handle>
);
```

For the request identified by <*request_handle*>, the function `nci_request_dump()` returns the information about the current state, severity, and each variable's name, type, and value. For example, for three variables, eleven pieces of information are returned: current state, severity, and name, type, and value of each variable. The information is returned as the value of the function. If there is an error, an empty array is returned.

5.5.15 `nci_request_info` *Function*

```
Result
nci_request_info(
    CDU <request_handle>,
    DU& <statename>,
    DU& <severity_name>,
    int*<severity>
);
```

For the request identified by <*request_handle*>, the function `nci_request_info()` returns the name of state in <*statename*>, severity string in <*severity_name*>, and severity value in the integer pointed to by <*severity*>. The function returns TRUE if the request exists, FALSE otherwise.

5.5.16 `nci_request_list` *Function*

```
Array(DU)
nci_request_list(
    CDU <mis_name> = null_du
);
```

The function `nci_request_list()` returns an array of request handles. The handles are a concatenation of the request name and the `topoNodeId`. `nci_parse_handle()` can be used to return the template name and `topoNodeId`.

5.5.17 `nci_request_start` *Function*

```
DU
nci_request_start(
    const char*<template_name>,
    CDU <topoNodeId>,
    const char*<handle>
);
```

The function `nci_request_start()` starts a management request with <*template_name*> against the managed object associated with `<topoNodeId>` in the topology part of the MIT. If the request starts successfully, the request handle is returned. This handle can be used, in `nci_request_delete()`, below, to stop and thereby delete the running request. These request calls are unconfirmed and asynchronous. If the request fails to start, a null_du is returned.

5.5.17.1 Alternative Syntax1

```
Array(DU)
nci_request_start(
    const char*<name>,
    const Array(DU)&<toponode_id_array>,
    const char*<handle>
);
```

This function call performs the same job as the one above, except that multiple requests using the same template can be started against multiple `topoNodes`. One request is launched for each topoNode Id in the array. The NCI obtains the objectInstance from the topoNode. An array of handles is returned. Each handle in the array corresponds to a topoNode in the same index in the <*toponode_id_array*>.

5.5.17.2 Alternative Syntax2

```
DU
nci_request_start(
    const char*<template_name>,
    CDU <oi>,
    CDU <topoNodeId>,
    const char*<handle>
);
```

5.5.17.3 Alternative Syntax3

```
DU
nci_request_start(
    const char*<template_name>,
    const Array(DU)&<oiSet>,
    CDU <topoNodeId>,
    const char*<handle>
);
```

Performs the same job as the preceding function, except this function handles multiple object instances.

5.5.17.4 Alternative Syntax4

```
Array(DU)
nci_request_start(
    const char*<template_name>,
    const Array(DU)&<oi_array>,
    const Array (DU)&<toponode_id_array>,
    const char*<handle>
);
```

These function calls are analogous to the two preceding calls, except that they start requests against any managed object (or objects, for the second call), even if there no `TopoNode` corresponds to that object. These request calls are unconfirmed and asynchronous. Returned values are as described for their `TopoNode` counterparts.

The DU is the ObjectInstance in either absolute-pathname or FDN-name format. The DU is the value of `$pollfdn` when the request is created. The second version, above, differs from the first in that it allows you to launch one request for each DU in the array.

An array of handles is returned. Each handle in the array corresponds to a topoNode in the same index in the <*toponode_id_array*>. If a given request failed to start, the handle in the returned array is a `null_du`.

5.5.17.5 Alternative Syntax5

```
Array(DU)
nci_request_start(
    const char*<template_name>,
    const Array(Array(DU))&<oiSetArray>,
    const Array(DU)&<toponode_id_array>,
    const char*<handle>
);
```

Performs the same job as the preceding function, except this function handles multiple object instances.

5.5.18 nci_severity_add Function

```
Result
nci_severity_add(
    const char*<severity_name>,
    int <severity>,
    const char*<color>,
    CDU <mis_name> = null_du
);
```

The function `nci_severity_add()` creates a new severity level in the MIS. The severity level can be from 1 to 32000 and the color is any X11 color such as red, purple, or yellow.

5.5.19 nci_severity_delete Function

```
Result
nci_severity_delete(
    char*<severity_name>,
    CDU <mis_name> = null_du
);
```

The function `nci_severity_delete()` deletes an existing severity in the MIS. The severity is deleted by name.

5.5.20 nci_state_add Function

```
Result
nci_state_add(
    NC_Defn&<def>,
    const char*<state_name>,
    const char*<pollrate_name>,
    const char*<severity_name>,
    const char*<state_desc>
);
```

The function `nci_state_add()` adds a state with name <*state_name*>, poll rate <*pollrate_name*>, severity <*severity_name*>, and description <*state_desc*> to the template identified by <*def*>. The function returns TRUE if the state is added, FALSE otherwise.

5.5.21 nci_state_delete Function

```
Result
nci_state_delete(
    NC)Defn&<def>,
    const char*<state_name>
);
```

The function `nci_state_delete()` deletes the state with name <*state_name*> from the template identified by <*def*>. The function returns TRUE if the state is deleted, FALSE otherwise.

5.5.22 `nci_state_get` *Function*

```
NC_State
nci_state_get(
    NC_Defn&<def>,
    const char*<state_name>
);
```

The function `nci_state_get()` gets the handle to identify the state with name <*state_name*> in the template identified by <*def*>. The function returns a valid `NC_State`, which is a handle on the Nerve Center state, and returns a default(invalid) `NC_State` if there is an error.

5.5.23 `nci_template_add` *Function*

```
Result
nci_template_add(
    NC_Defn&<def>,
    const char*<name>,
    const char*<descr>
    CDU <mis_name> = null_du
);
```

The function `nci_template_add`() is used when you have updated an existing request template. The first time a template is created you use `nci_template_store()`. Subsequent edits require `nci_template_add()`. This function cannot be used to modify a template's name. An alternative is to use the function `nci_template_copy()` to copy the template using a different name and then delete the original template.

5.5.24 nci_template_copy Function

```
NC_Defn
nci_template_copy(
    NC_Defn&<source_defn>,
    const char*<newname>,
    const char*<descr>,
    CDU <mis_name> = null_du
);
```

The function `nci_template_copy()` returns a handle for a new copy of template name. The old handle of the template must be passed in.

5.5.25 nci_template_create Function

```
NC_Defn
nci_template_create(
    const char*<template_name>,
    const char*<template_desc>,
    CDU <mis_name> = null_du
);
```

The function `nci_template_create()` creates a handle for a request template. After creation, the template can be built with other NCI library calls passing `NC_Defn` as the handle for the template.

5.5.26 nci_templete_delete Function

```
Result
nci_template_delete(
    NC_Defn&<def>
);
```

The function `nci_template_delete()` deletes an existing request template from the MIS. This call fails if any requests are currently using this template

5.5.27 nci_templae_find *Function*

```
NC_Defn
nci_template_find(
    const char*<template_name>,
    CDU <mis_name> = null_du
);
```

The function `nci_template_find()` is used to get the handle for an existing request for editing.

5.5.28 nci_template_revert *Function*

```
Result
nci_template_store(
    NC_Defn&<def>,
    const char*<template_name>,
    const char*<descr>,
    CDU <mis_name> = null_du
);
```

The function nci_template_revert()allows the user to undo any changes that may have been made to the template specified by <template_name>, but not saved in the MIS. This function is effective only if a nci_template_store() has not yet been invoked on the given template. Typically, the usage would be to get the template definition by invoking nci_template_find(), do changes to the template definition like adding states, conditions, and transitions, and then invoke nci_template_revert() to undo the changes before any call to nci_template_store() is made.

5.5.29 nci_template_store Function

```
Result
nci_template_store(
    NC_Defn&<def>,
    const char*<template_name>,
    const char*<descr>,
    CDU <mis_name> = null_du
);
```

The function nci_template_store() is called after the request template has been completely built. If nci_template_store() returns TRUE, the request template is ready for event management.

5.5.30 nci_transition_add Function

```
Result
nci_transition_add(
    NC_Defn&<def>,
    const char*<from>,
    const char*<to>,
    const char*<condition>,
    const char*<action>,
    const char*<arg0>,
    const char*<arg1>
);
```

The function nci_transition_add() adds a transition to an existing template. The transition must have “from” and “to” states and the condition must exist or the call fails. Three possible actions are supported: unixcmd, condition, and mail. If unix command or mail actions are passed, there must be a double-quoted *<arg1>*.

For more information on transitions, refer to the “Request Designer” chapter in the *Solstice Enterprise Manager Reference Manual*.

5.5.31 `nci_transition_delete` *Function*

```
Result
nci_transition_delete(
    NC_Defn&<def>,
    const char*<from_state>,
    const char*<to_state>,
    const char*<condition>,
    const char*<action>,
    const char*<arg0>,
    const char*<arg1>
);
```

The function `nci_transition_delete()` deletes a transition between two states.

5.5.32 `nci_transition_find` *Function*

```
NC_Transition
nci_transition_find(
    NC_Defn&<def>,
    const char*<from_state>,
    const char*<to_state>,
    const char*<condition>
);
```

The function `nci_transition_find()` gets the transition in template identified by <*def*> going from the state <*from_state*> to <*to_state*> on the condition <*condition*>. The function returns a valid `NC_Transition`, which is a handle on the Nerve Center state-transition, and returns a default (invalid) `NC_Transition` if there is an error.

5.5.33 nci_transition_get Function

```
NC_Transition
nci_transition_get(
    NC_State&<from_state>,
    NC_State&<to_state>,
    const char*<condition>
);
```

5.5.33.1 Description

The function `nci_transition_get()` gets the handle on transition from the state identified by <*from_state*> to the state identified by <*to_state*> on the condition <*condition*>. The function returns a valid NC_Transition, which is a handle on the Nerve Center state-transition, and returns a default (invalid) NC_Transition if there is an error.

5.6 Event Request Example

This program creates pollrates, severities, and conditions, and then uses them to define a request template for managing an SNMP host.

Code Example 5-1 Sample Event Request

```
#include <hi.hh>
#include <stdlib.h>
#include <sys/types.h>
#include “error.hh”
#include “nc_api.hh”
#include “nc_def.hh”
#include “nc_coll.hh”

Error   error;
Exception *exception;
ExceptionValue cur_except;

void create_pollrates();
void create_severities();
void create_conditions();
void create_template(char*);
void fail(char*);
```

Code Example 5-1 Sample Event Request

```
NC_Defn nc_handle;

main(int argc, char**argv)
{
    if (argc != 2) {
       printf(“Usage: create_template <template_name>\n”);
       exit(1);
    }
    DU error_msg;
    if (!nci_init(“localhost”, error_msg))  {
        printf(“libnci initialization failed:%s/n”, error_msg.chp());
        exit(1);
    }
    create_pollrates();
    create_severities();
    create_conditions();
    create_template(argv[1]);
}

void
create_pollrates()
{
           if (!nci_pollrate_add(“FastPoll”,20))
          fail(“nci_pollrate_add”);
        if (!nci_pollrate_add(“SlowPoll”,60))
          fail(“nci_pollrate_add”);
}

void
create_severities()
{
        if (!nci_severity_add(“down”, 18, “red”))
          fail(“nci_severity_add”);
        if (!nci_severity_add(“ok”, 16, “green”))
          fail(“nci_severity_add”);
}

void
create_conditions()
{
    const char *SYS = “$tmp = \”/internetClassId=\
{1 3 6 1 4 1 42 3 2 3 1 1 3 6 1 2 1 1 0}\”;\n$pollfdn = append_rdn($pollfdn,$tmp);\ntrue;”;
```

Code Example 5-1 Sample Event Request

```
          if (!nci_condition_add(“SetSystem”, “Set the polling fdn”,SYS))
            fail(“nci_condition_add”);
          if (!nci_condition_add(“IsSystemDescr”, “Poll for System Description”,
                            “defined(&sysDescr);”))
            fail(“nci_condition_add”);
          if (!nci_condition_add(“IsNotSystemDescr”, “If Can’t Reach the System”,
                         “NOT defined(&sysDescr);”))
            fail(“nci_condition_add”);
          if (!nci_condition_add(“UndefineSystemDescr”,
              “Undefine the System Description”,”undefine(&sysDescr);”))
            fail(“nci_condition_add”);
}

void
create_template(
    char *name
)
{
    NC_Defn nc = nci_template_create(name,”Test Template”);
    if (!nc)
       fail(“nci_template_create”);

    // Adding States
    if (!nci_state_add(nc,”Init”,”Poll”,”Normal”,”Initialization State”))
       fail(“nci_state_add”);
    if (!nci_state_add(nc,”Poll”,”Poll”,”Normal”,”Polling State”))
       fail(“nci_state_add”);
    if (!nci_state_add(nc,”Up”,”Poll”,”ok”,”System Up”)))
       fail(“nci_state_add”);
    if (!nci_state_add(nc,”Down”,”Poll”,”down”,”System Down”))
       fail(“nci_state_add”);

    // Add Transitions
    if (!nci_transition_add(nc,”Init”,”Poll”,”SetSystem”,NULL,NULL,NULL))
       fail(“nci_transition_add”);
    if (!nci_transition_add(nc,”Poll”,”Up”,”IsSystemDescr”,
               “CONDITION”,”UndefineSystemDescr”,NULL))
       fail(“nci_transition_add”);
    if (!nci_transition_add(nc,”Poll”,”Down”,
          “IsNotSystemDescr”,NULL,NULL,NULL))
       fail(“nci_transition_add”);
```

Code Example 5-1 Sample Event Request

```
    if (!nci_transition_add(nc,”Up”,”Down”,”IsNotSystemDescr”,NULL,NULL,NULL))
       fail(“nci_transition_add”);
    if (!nci_transition_add(nc,”Up”,”Up”,”IsSystemDescr”,
           “CONDITION”,”UndefineSystemDescr”,NULL))
       fail(“nci_transition_add”);
    if (!nci_transition_add(nc,”Down”,”Up”,”IsSystemDescr”,
           “CONDITION”,”UndefineSystemDescr”,NULL))
       fail(“nci_transition_add”);

        if (!nci_template_store(nc, name, “Def”))
          fail(“nci_template_store”);
}

void
fail(char *s)
{
    printf(“%s: Failed exiting.\”, s);
    exit(1);
}
```

Object Services API 6

The object development tools (ODT) allow developers who implement object classes to describe how and when a notice is sent after receiving an event. This API is useful for developing manager and agent network management applications. Developers can use it to customize behaviors for the GDMO classes, rather than accept the default behaviors.

The ODT do not allow users to extend or override any existing services or object behavior within the MIS, such as the Topology database, NerveCenter, logging, or Event Service.

6.1 Operational Flow

The Services API provides a set of programming interfaces that can be used by an application developer when writing object behavior software. An application developer is not required to use the services functions. However, these functions make it easier to perform some common tasks related to inter-object communication from within.

The operational flow of a request issued using the Services API is based on the MIS `Message` and `MessageSAP` C++ classes on object behavior. The `Message` and `MessageSAP` C++ classes are used throughout the MIS and are also the basis for the low level PMI interface. The Services API hides the `Message` classes and the `MessageSAP` classes used by the low level PMI. The classes are hidden primarily to simplify this API and also for the following reasons:

- Most messages sent through this API can use a number of default values. The Services API function calls provide default values for all parameters not specifically required for a particular operation.
- Only a single well defined `MessageSAP` is required for these functions. The services function calls all involve communication between the object access module (OAM) and the message routing module (MRM). The OAM contains both user-developed object behavior code and the generated object behavior code. The MRM handles routing for all message requests and responses.

6.2 Service Request Function Parameters

Table 6-1 gives a detailed description of the service request function parameters defined for the Services API.

Table 6-1 Service Request Function Parameters Table

Parameter	**Description**
`const Asn1Value oc`	Instance of the Asn1Value C++ class that contains an Object Identifier (OID) for a managed object class. For each of the Service API request functions, except the `send_event_req` function, this parameter contains the OID for Actual Class. Actual Class is an ISO defined OID that matches the class of any managed object on which an operation is performed. The `oc` parameter is used as follows: *Single Object Selection (Base Object Only Scoping):* The OID specifies the object class of the managed object from which attribute values are retrieved, using the Get operation. *Multiple Object Selection Using Scoping and Filtering:* The OID parameter specifies the object class of the managed object used as the starting point for the selection of managed objects from which attribute values are retrieved using the Get operation. The CMIS and CMIP specifications refer to this parameter as either the base object class or managed object class, depending on the type of operation being performed.

Table 6-1 Service Request Function Parameters Table

Parameter	**Description**
`const Asn1Value oi`	Instance of the Asn1Value C++ class that contains either a distinguished name (context specific 2) or a local distinguished name (context specific 4) for a managed object instance. The `oi` parameter is used as follows: *Single Object Selection (Base Object Only Scoping):* The `oi` parameter specifies the name of the managed object instance from which the request operation is performed. *Multiple Object Selection Using Scoping and Filtering:* The `oi` parameter specifies the name of the managed object instance to be used as the starting point for the selection of managed objects on which the request operation is performed. The CMIS and CMIP specifications refer to this parameter as either the base object class or managed object instance, depending on the type of operation being performed. Specify this parameter as a null Asn1Value (Asn1Value()) if a managed object instance name is specified by the `superior_oi` parameter.
`const Callback cb`	Instance of the Callback C++ class, which can contain two pointers: the first is a pointer of a callback function to be invoked when a response to an operation is received; the second is a pointer to application developer-specified data (commonly referred to as user data) to be passed to the callback function when it is invoked. The user data pointer is always optional. The `cb` parameter is required or optional for service request functions that support it. If specified, a *confirmed* service request is issued. If not specified, an *unconfirmed* service request is issued.
`const Asn1Value attr_list`	Instance of the Asn1Value C++ class that contains a set of OIDs for attributes. Attributes are normally members of the object class specified by the `oc` parameter or members of the class of an object instance identified within a scoped operation.
`const Asn1Value modify_list`	The `modify_list` parameter is a set of attribute ID and attribute value pairs that specify which attributes are to be modified for a `send_set_req` service request operation and also specify the new values for the attributes. The `modify_list` parameter is an instance of the Ans1Value C++ class and is typically a sequence of an OID that identifies an attribute, followed by a value for the attribute.
`const Asn1Value action_type`	Instance of the Asn1Value C++ class that contains an object identifier (OID) that specifies the type of action generated by the `send_action_req` function.
`const Asn1Value action_info`	Instance of the Asn1Value C++ class that contains any event information associated with the type of action specified by the `action_type` parameter. The `action_info` parameter typically contains a sequence or set of ASN.1-defined values. The `action_info` parameter is optional for the `send_action_req` service function but must be present if a `WITH INFORMATION SYNTAX` construct is specified as part of the GDMO definition for the action type specified by the `action_type` parameter.

Table 6-1 Service Request Function Parameters Table

Parameter	Description
`const Asn1Value attr_value_list`	A set of attribute ID and attribute value pairs that specify the attributes assigned new values by a `send_create_req` service request operation. The values specified in the `send_create_req` service function override the corresponding attributes from the reference object (if specified using the `reference_oi` parameter) or from the default value specified in the GDMO definition for the managed object class. The `attr_value_list` parameter is an instance of the Asn1Value C++ class and is typically a sequence of an OID that identifies an attribute, followed by a value for the attribute. The `attr_value_list` parameter is an optional parameter for the `send_create_req` service request, although values must be specified for all mandatory attributes defined in the GDMO managed object class definition for which an instance is being created. In other words, the mandatory attribute values must be specified in the GDMO definition in a default value clause, supplied from a reference object, or else specified in the `attr_value_list`.
`const Asn1Value superior_oi`	An instance of the Asn1Value C++ class, this parameter is used only with the `send_create_req` function. The parameter contains either a distinguished name (context specific 2) or a local distinguished name (context specific 4) for an existing managed object instance that is to be the superior—in the MIT—of the managed object instance created. This parameter should not be specified or should be specified as a null Asn1Value, if a managed object instance name is specified by the `oi` parameter.
`const Asn1Value reference_oi`	An instance of the Asn1Value C++ class, this parameter is used only with the `send_create_req` function. It contains either a distinguished name (context specific 2) or a local distinguished name (context specific 4) for an existing managed object instance that is of the same class as the managed object instance created. Attribute values associated with the managed object specified by the `reference_oi` parameter become default values for those attributes not specified by the `attr_value_list` parameter of the `send_create_req` function.
`const Asn1Value event_type`	Instance of the Asn1Value C++ class that contains an OID specifying the type of notification to be generated by the `send_event_req` function.
`const Asn1Value event_info`	An Asn1Value C++ class that contains any event information associated with the type of Notification specified by the `event_type` parameter. This parameter typically contains a sequence or set of ASN.1-defined values and is optional for the `send_event_req` service function. It must be present if a `WITH INFORMATION SYNTAX` construct is specified as part of the GDMO definition for the notification type specified by the `event_type` parameter.
`const Asn1Value event_time`	Instance of the Asn1Value C++ class containing a value for the time at which a notification is generated.

Table 6-1 Service Request Function Parameters Table

Parameter	**Description**
`MessId id`	Identifier that uniquely identifies a service request operation. If specified, the value for this parameter is generated and set by the service request function.
`const MessScope scope`	Specifies the type of scoping used for a service request operation. The possible types of scoping are `BASE_OBJECT`, `NTH_LEVEL`, `BASE-TO-NTH_Level`, and `ALL_LEVELS`. Optional for all service request functions that support it. If not specified, it defaults to `BASE_OBJECT`.
`const Asn1Value filter`	Instance of the Asn1Value C++ class that contains a CMISFilter (refer to the ISO DMI for a definition of the CMIS filter). All objects selected by the scoping parameter are tested against a filter. The service request operation is performed only on those objects that pass the filter test. Optional parameter for all service request functions that support it. If not specified, it defaults to a filter that matches all managed objects selected by the `scope` parameter.
`const MessSync sync`	Specifies the type of synchronization used for a service request operation. It is an enumerated type and can take on the value `ATOMIC` or `BEST_EFFORT`. Optional parameter for all service request functions that support it. If not specified, it defaults to `BEST_EFFORT`. The use of `ATOMIC` is rarely supported by remote objects.
`const Asn1Value access`	Reserved parameter not normally specified. The `access` parameter defines the access control that objects selected for a service request operation must pass. The service request operation is not performed by any managed object that does not pass the access control.

6.3 *Service Response Callback Function Parameters*

For a description of the parameters, see Table 6-2.

Table 6-2 Service Response Callback Function Parameter Table

Parameter	**Description**
`Ptr userdata`	Optional parameter intended for all service response callback functions. It is a `void *` pointer to data specified by the application developer in the callback parameter of a service request function.
`Ptr message`	Mandatory parameter for all service response callback functions. It is a `void *` pointer to the message generated in response to a service request function.

6.4 Services Interface Descriptions and Examples

The following service request functions and service indication functions are supported by the Services API. In general, the services interfaces specified here contain both mandatory and optional parameters. Mandatory parameters are typically passed by reference. Optional parameters are typically passed either by value or by a pointer to a value. Mandatory parameters are ordered prior to the optional parameters for each function. Optional parameters for each function are ordered by placing the most-likely-to-be-specified optional parameters first.

This section includes the following functions:

- *Get Request Service*
- *Get Response Callback*
- *Set Request Service*
- *Set Response Callback*
- *Action Request Service*
- *Action Response Callback*
- *Create Request Service*
- *Create Response Callback*
- *Delete Request Service*
- *Delete Response Callback*
- *Delete Response Callback Parameter Description*
- *Event Report Request Service (Unconfirmed)*
- *Event Report Response Callback*

6.4.1 Get Request Service

The following subsections explain the get request service.

6.4.1.1 *Interface Signature*

```
Result send_get_req(
    const Asn1Value &oc,
    const Asn1Value &oi,
    const Callback &cb,                   // Always confirmed
    const Asn1Value attr_list = Asn1Value(), // Default: Get all
                                            attributes
    MessId *id =0,
    const MessScope scope = MessScope(),//Default: Base object
only
    const Asn1Value filter = Asn1Value(),  //Default: Matches all
    const MessSync sync = BEST_EFFORT,
    const Asn1Value access = Asn1Value()); //Default: No access
cntrl
```

6.4.1.2 `send_get_request` *Parameter Descriptions*

Table 6-3 shows the `send_get_request` parameters.

Table 6-3 send_get_request Parameter Table

Parameter	**Description**	**Required/ Optional**
`const Asn1Value oc`	Specifies the class of the base managed object.	Required
`const Asn1Value oi`	Distinguished name of local distinguished name for the base managed object.	Required
`const Callback cb`	Specifies the name of a callback function invoked when a response to the get request is received and can be used to specify user data that is passed to the callback function when it is invoked.	Required
`const Asn1Value attr_list`	List of attribute OIDs whose attribute values are to be returned in response to the get request operation. If not specified, all attributes defined for the managed object class are returned in the response.	Optional
`MessId id`	Provides a unique identifier for a particular Get request operation. If specified, the value for this parameter is generated and set by the `send_get_req` function.	Optional
`const MessScope scope`	Defines the type of scoping used for this request operation. If not specified, it defaults to `BASE_OBJECT_ONLY`.	Optional

Table 6-3 send_get_request Parameter Table

Parameter	Description	Required/ Optional
`const Asn1Value filter`	Defines a filter to be passed by all objects selected by the scoping parameter. If not specified, it defaults to a filter that matches all managed objects selected by the scope parameter.	Optional
`const MessSync sync`	Defines the type of synchronization used for this request operation. If not specified, it defaults to BEST_EFFORT.	Optional
`const Asn1Value access`	Reserved parameter that should not be used at this time.	Not available

6.4.1.3 send_get_request Examples

Get Request: Base Object Only

Code Example 6-1 Base Object Only Example

```
Result
get_system_object_attributes(char *system_name)
{
    Asn1Value actual_oc;
    Asn1Value system_oi;

    //********************************************
    // (Confirmed) Get request
    //********************************************
    // Send a get request to an instance of the
    // system managed object class. The system
    // the request is sent to is identified by
    // the system_name parameter.

    VTRY {

    // Encode oc OID. In this example, actualClass is used
    // instead of the OID for the system managed object
    // class.
    // Note: CMIP requires a TAG_CONT(0) encoding for the
    // object class rather than TAG_OID (refer to x711.asn1
    // for encoding spec).

        TTRYRES(actual_oc.encode_oid(TAG_CONT(0),
            Oid("2.9.3.4.3.42")));
```

Code Example 6-1 Base Object Only Example

```
Result
    // Encode the distinguished name for an instance of the
    // system managed object class. Either the distinguished
    // name form (TAG_CONT(2)) or the local distinguished name
    // form (TAG_CONT(4)) can be used with the get operation.
    // The distinguished name form is used in this example.
    // The get_sys_dn function is included in Section 6.5.

        TTRYRES(get_sys_dn(system_name, system_oi));

    // In this example, all attributes from the system object
class
    // are retrieved. No attribute ID list is required.

    // Send the get request (Always Confirmed). No user data is
    // specified for the callback parameter. The get_req_cb
function
    // is included as part of the Services API examples.

         objsvc_test.print("About to issue Get Request\n");
         if (send_get_req(actual_oc, system_oi,
             Callback(get_req_cb, 0)) != OK) {
             objsvc_test.print("Error issuing Get Request\n");
            return NOT_OK;
        }
        else
            objsvc_test.print("Issued Get Request\n");
    }
    VBEGHANDLERS
    VCATCHALL {
        objsvc_test.print("\nError encoding Get Request\n");
        return NOT_OK;
    }
    VENDHANDLERS
    return OK;
}
```

Get Request: Scoped Operation

Code Example 6-2 Scoped Operation Example

```
Result
get_application_attributes(char *system_name)
{
    Asn1Value actual_oc;
    Asn1Value system_oi;
    Asn1Value em_mis_rdn;

    //*************************************
    // (Confirmed) Scoped get request
    //*************************************
    // Get the emApplicationID, emApplicationType, and the
    // emUserID attribute values for each application instance
    // object under an emKernel object. The emKernel object
    // is located under the system object in the MIT.

    VTRY {

    // Encode oc OID. In this example, actualClass is used
    // instead of the OID for the system managed object
    // class.
    // Note: CMIP requires a TAG_CONT(0) encoding for the
    // object class rather than TAG_OID (refer to x711.asn1
    // for encoding spec).

        TTRYRES(actual_oc.encode_oid(TAG_CONT(0),
            Oid("2.9.3.4.3.42")));

    // Encode the distinguished name for an instance of the
    // emKernel managed object class. Either the distinguished
    // name form (TAG_CONT(2)) or the local distinguished name
    // form (TAG_CONT(4)) can be used with the get operation.
    // The distinguished name form is used in this example.
    // The get_sys_dn and get_graphstr_rdn functions are included
    // in Section 6.5.

        TTRYRES(get_sys_dn(system_name, system_oi));
        TTRYRES(get_graphstr_rdn(
            "2.9.3.5.7.11","EM-MIS", em_mis_rdn));
        TTRYRES(system_oi.add_component(em_mis_rdn));

    // Encode the attribute list. The CMIP spec specifies
    // TAG_CONT(12) as the tag for the attribute list.
```

Code Example 6-2 Scoped Operation Example

```
Result

        Asn1Value attrlist;
        Asn1Value enc_oid1, enc_oid2, enc_oid6;
        Oid oid1("1.3.6.1.4.1.42.2.2.2.1.7.1"); //
emApplicationID
        Oid oid2("1.3.6.1.4.1.42.2.2.2.1.7.2"); //
emApplicationType
        Oid oid6("1.3.6.1.4.1.42.2.2.2.1.7.6"); // emUserID
        TTRYRES(attrlist.start_construct(TAG_CONT(12)));
        TTRYRES(enc_oid1.encode_oid(TAG_CONT(0), oid1));
        TTRYRES(enc_oid2.encode_oid(TAG_CONT(0), oid2));
        TTRYRES(enc_oid6.encode_oid(TAG_CONT(0), oid6));
        TTRYRES(attrlist.add_component(enc_oid1));
        TTRYRES(attrlist.add_component(enc_oid2));
        TTRYRES(attrlist.add_component(enc_oid6));

    // Send the scoped get request (Always Confirmed). No user
data
    // is specified for the callback parameter. The
    // scoped_get_req_cb function is included as part of the
    // Services API examples.

        objsvc_test.print("About to issue Scoped Get Request\n");
         if (send_get_req(actual_oc, system_oi,
             Callback(scoped_get_req_cb, 0), attrlist,
            0, MessScope(NTH_LEVEL, 1)) != OK) {
             objsvc_test.print("Error issuing Scoped Get
Request\n");
            return NOT_OK;
        }
        else
            objsvc_test.print("Issued Scoped Get Request\n");
    }
    VBEGHANDLERS
    VCATCHALL {
        objsvc_test.print("\nError encoding Get Request\n");
        return NOT_OK;
    }
    VENDHANDLERS
    return OK;
}
```

6.4.2 Get Response Callback

The send_get_req service function requires a callback function. The name of the callback function provided by the application developer must match the name of the callback function specified in the Callback parameter of the send_get_req function.

6.4.2.1 Interface Signature

void *user-provided-get-response-callback*(

```
    Ptr userdata,                  // Pointer to user supplied
 data

    Ptr message);                  // Pointer to GetRes message
```

6.4.2.2 Get Response Callback Parameter Descriptions

The following table shows the Get response callback parameters.

Table 6-4 Get Response Callback Parameter Table

Parameter	**Description**
Ptr userdata	A void * pointer to data specified by the application developer in the callback parameter of the send_get_req function.
Ptr message	A void * pointer to the GetRes message generated in response to the send_get_req function.

6.4.2.3 Get Response Callback Examples

Callback Function: Single Response (Base Object Only)

Code Example 6-3 Callback Function, Single Response Example

```
void
get_req_cb(Ptr, Ptr get_response_msg)
{
    objsvc_test.print("Get Request callback\n");
    Message *resp = (Message *)get_response_msg;

    VTRY {
        if ( resp->type() == GET_RES)  {
            GetRes *g_resp = (GetRes *)resp;
            g_resp->print(objsvc_test);

        // The Asn1Value decoding functions, including the
        // member functions get_first_component and
        // get_next_component can be used to examine the member
data
        // included in the response message at this point

        //  ...

        }
        else if ( resp->is_error() ) {
            objsvc_test.print("Get response error received\n");
            objsvc_test.print( "message type = %s\n",
                MessType_fmt(resp->type()));
            resp->print(objsvc_error);
        }
        else {
            objsvc_test.print(
                "Unexpected or invalid response received\n");
            objsvc_test.print( "message type = %s\n",
                MessType_fmt(resp->type()));
            resp->print(objsvc_error);
        }
    }
    VBEGHANDLERS
    VCATCHALL {
        objsvc_test.print("Error processing response for Get\n");
        resp->print(objsvc_error);
    }
```

Code Example 6-3 Callback Function, Single Response Example

```
void
    VENDHANDLERS
    if ( resp )
        Message:: delete_message(resp);
}
```

Callback Function: Multiple Responses (Scoped Operation)

Code Example 6-4 Callback Function, Multiple Responses Example

```
static int resp_count = 0;
static int err_count = 0;
static int unknown_count = 0;
void
sg_req_cb(Ptr userdata, Ptr calldata)
{
    objsvc_test.print("Scoped Get Request callback:");
    Message *resp = (Message *)calldata;

    VTRY {
        if ( resp->type() == GET_RES)  {
            GetRes *sg_resp = (GetRes *)resp;
            if (sg_resp->linked) {
                objsvc_test.print("*** LINKED Response ***\n");
                resp_count++;
                sg_resp->print(objsvc_test);

            // The Asn1Value decoding functions, including the
            // member functions get_first_component and
            // get_next_component can be used to examine the
member
            // data included in the response message here.

            //  ...

            } else {
                objsvc_test.print("**** Final Response ****\n");
                sg_resp->print(objsvc_test);
                objsvc_test.print("Valid: %d, Error: %d,Invalid:
%d\n",
                resp_count, err_count, unknown_count);

            // Final response processing can be performed here.
The
```

Code Example 6-4 Callback Function, Multiple Responses Example

```
static int resp_count = 0;
            // final response message does not contain any
attribute
            // value data

            //  ...

            }
        }
        else if ( resp->is_error() ) {
            objsvc_test.print("Error Response\n");
            objsvc_test.print( "message type = %s\n",
                MessType_fmt(resp->type()));
            resp->print(objsvc_error);
            err_count++;
        }
        else {
            objsvc_test.print("Invalid Message\n");
            objsvc_test.print( "message type = %s\n",
                MessType_fmt(resp->type()));
            resp->print(objsvc_error);
            unknown_count++;
        }
    }
    VBEGHANDLERS
    VCATCHALL {
        objsvc_test.print("\nError processing response for Scoped
Get\n");
    }
    VENDHANDLERS

    if ( resp )
        Message:: delete_message(resp);
}
```

6.4.3 Set Request Service

This section includes information on the interface signature, `send_set_request` parameter, and an example.

6.4.3.1 Interface Signature

Code Example 6-5 Interface Signature Example

```
Result send_set_req(
    const Asn1Value &oc,
    const Asn1Value &oi,
    const Asn1Value &modify_list,
    const Callback cb = 0,                // Default: Unconfirmed
    MessId *id = 0,
    const MessScope scope = MessScope(), // Default: Is base
object
        only
    const Asn1Value filter = Asn1Value(), // Default: Matches all
    const MessSync sync = BEST_EFFORT,
    const Asn1Value access = Asn1Value()); // Default: No access
        cntrl
```

6.4.3.2 `send_set_request` Parameter Descriptions

See Table 6-5 for these parameter descriptions.

Table 6-5 `send_set_request` Parameter Descriptions

Parameter	Description	Required/ Optional
`const Asn1Value oc`	OID that specifies the class of the base managed object.	Required
`const Asn1Value oi`	Distinguished name or local distinguished name for the base managed object.	Required
`const Ans1Value modify_list`	List of attribute ID and attribute value pairs that specify which attributes are to be modified by the set request operation and what the new values are for the attributes.	Required
`const Callback cb`	Specifies the name of a callback function invoked when a response to the set request operation is received. Can also be used to specify user data passed to the callback function when it is invoked. If specified, a confirmed set request operation is issued and the callback function is invoked when the response is received. If not specified, an unconfirmed set request operation is issued and no response is generated.	Optional
`MessId id`	Provides a unique identifier for a particular set request operation. If specified, the value for this parameter is generated and set by the `send_set_req` function.	Optional

Table 6-5 `send_set_request` Parameter Descriptions

Parameter	**Description**	**Required/ Optional**
`const MessScope scope`	Defines the type of scoping used for this request operation. If not specified, it defaults to BASE_OBJECT_ONLY.	Optional
`const Asn1Value filter`	Defines a filter to be passed by all objects selected by the scoping parameter. If not specified, it defaults to a filter that matches all managed objects selected by the scope parameter.	Optional
`const MessSync sync`	Defines the type of synchronization used for this request operation. If not specified, it defaults to BEST_EFFORT.	Optional
`const Asn1Value access`	This parameter is reserved at this time.	Not available

6.4.3.3 `send_set_request` *Example*

Code Example 6-6 `send_set_request` Example

```
Result
set_log_adminState(char *system_name)
{
    Asn1Value actual_oc;
    Asn1Value log_fdn;
    Asn1Value log_rdn;
    Asn1Value mod_list;

    //*******************************************
    // Confirmed Set request
    //*******************************************
    // Send a confirmed set request to an instance of the
    // log managed object class. The log that
    // the request is sent to is
    // contained under the system identified by
    // the system_name parameter.

    VTRY {

    // Encode oc OID. In this example, actualClass is used
    // instead of the OID for the log managed object
    // class.
    // Note: CMIP requires a TAG_CONT(0) encoding for the
    // object class rather than TAG_OID (refer to x711.asn1
    // for encoding spec).
```

Code Example 6-6 `send_set_request` Example

```
Result
       TTRYRES(actual_oc.encode_oid(TAG_CONT(0),
           Oid("2.9.3.4.3.42")));

    // Encode the distinguished name for an instance of the
    // log managed object class. The instance logId="AlarmLog"
    // is used here. Either the distinguished name form
    // (TAG_CONT(2)) or the local distinguished name form
    // (TAG_CONT(4)) can be used with the set operation. The
    // distinguished name form is used in this example.
    // The get_sys_dn and get_graphstr_rdn functions are included
    // in Section 6.5.

        TTRYRES(get_sys_dn(system_name, log_fdn));
        TTRYRES(get_graphstr_rdn(Oid("2.9.3.2.7.2"), "AlarmLog",
            log_rdn));
        TTRYRES(log_fdn.add_component(log_rdn));

    // Encode the modification list for the set operation.
    // The administrativeState of the log object is set to
    // locked in this example.
    // The CMIP spec specifies TAG_CONT(12) as the tag for
    // the attribute modification list.

        TTRYRES(modlist.start_construct(TAG_CONT(12)));
        Asn1Value av;
        TTRYRES(av.encode_enum(TAG_ENUM, 0)); // Set to LOCKED
        Asn1Value set;
        Asn1Value comp;
        Oid oid("2.9.3.2.7.31"); // Set administrativeState
        TTRYRES(set.start_construct(TAG_SEQ));
        TTRYRES(comp.encode_oid(TAG_CONT(0), oid));
        TTRYRES(set.add_component(comp));
        TTRYRES(set.add_component(av));
        TTRYRES(modlist.add_component(set));

    // Send the confirmed set request. No user data is
    // specified for the callback parameter. The set_req_cb
function
    // is included as part of the Services API examples.

        oamsvc_test.print("About to issue Confirmed Set
Request\n");
        if (send_set_req(actual_oc, log_fdn, modlist,
```

Code Example 6-6 `send_set_request` Example

```
Result
            Callback(set_req_cb, userdata)) != OK)
             objsvc_test.print("Error issuing Confirmed Set
Request\n");
            return NOT_OK;
        }
        else
            objsvc_test.print("Issued Confirmed Set Request\n");
    }
    VBEGHANDLERS
    VCATCHALL {
        objsvc_test.print("\nError encoding Set Request\n");
        return NOT_OK;
    }
    VENDHANDLERS
    return OK;
}
```

6.4.4 Set Response Callback

This callback function is used only in conjunction with confirmed Set requests. The name of the callback function must match the name of the callback function specified in the callback parameter of the `send_set_req` function.

6.4.4.1 *Interface Signature*

```
void user-provided-set-response-callback(
    Ptr userdata,                    // Pointer to user supplied
data
    Ptr message);                    // Pointer to SetRes message
```

6.4.4.2 *Set Response Callback Parameter Description*

See Table 6-6 for the description of this parameter.

Table 6-6 Set Response Callback Parameter Table

Parameter	**Description**
`Ptr userdata`	A `void *` pointer to data specified by the application developer in the callback parameter of the `send_set_req` function.
`Ptr message`	A `void *` pointer to the `SetRes` message generated in response to the `send_set_req` function.

6.4.4.3 *Set Response Callback Example*

Code Example 6-7 Set Response Example

```
void
set_req_cb(Ptr, Ptr set_response_msg)
{
    objsvc_test.print("Set Request callback\n");
    Message *resp = (Message *)set_response_msg;

    VTRY {
        if ( resp->type() == SET_RES)  {
            SetRes *s_resp = (SetRes *)resp;
            s_resp->print(objsvc_test);

        // The Asn1Value decoding functions, including the
        // member functions get_first_component and
        // get_next_component can be used to examine the member
data
        // included in the response message at this point

        //  ...

        }
        else if ( resp->is_error() ) {
            objsvc_test.print("Set response error received\n");
            objsvc_test.print( "message type = %s\n",
                MessType_fmt(resp->type()));
            resp->print(objsvc_error);
        }
        else {
            objsvc_test.print(
                "Unexpected or invalid response received\n");
            objsvc_test.print( "message type = %s\n",
                MessType_fmt(resp->type()));
            resp->print(objsvc_error);
        }
    }
    VBEGHANDLERS
    VCATCHALL {
        oamsvc_test.print("Error processing response for Set\n");
    }
    VENDHANDLERS
```

Code Example 6-7 Set Response Example

```
void
    if ( resp )
        Message:: delete_message(resp);
}
```

6.4.5 *Action Request Service*

6.4.5.1 *Interface Signature*

Code Example 6-8 Interface Signature for Action Request Service

```
Result send_action_req(
    const Asn1Value &oc,
    const Asn1Value &oi,
    const Asn1Value &action_type,
    const Asn1Value action_info = Asn1Value(), // Default: No
action
        Info
    const Callback cb = 0,            // Default: Unconfirmed
    MessId *id = 0,
    const MessScope scope = MessScope(), // Default: Base object
Only
    const Asn1Value filter = Asn1Value(), // Default: Matches all
    const MessSync sync = BEST_EFFORT,
    const Asn1Value access = Asn1Value()); // Default: No access
Cntrl
```

6.4.5.2 *send_action_req Parameter Descriptions*

Code Example 6-9 `send_action_req` Parameter Table

Parameter	**Description**	**Required/ Optional**
`const Asn1Value oc`	OID that specifies the class of the base managed object.	Required
`const Asn1Value oi`	Distinguished name or local distinguished name for the base managed object.	Required
`const Asn1Value action_type`	OID that specifies the type of action to be performed by the `send_action_req` function.	Required

Code Example 6-9 `send_action_req` Parameter Table

Parameter	**Description**	**Required/ Optional**
`const Asn1Value action_info`	This parameter contains information associated with the type of action specified by the `action_type` parameter. This option must be specified if a `WITH INFORMATION SYNTAX` construct is part of the GDMO definition for the action type specified by the `action_type` parameter.	Optional
`const Callback cb`	Specifies the name of a callback function invoked when a response to the action request operation is received and can be used to specify user data that is passed to the callback function when it is invoked. The callback parameter is optional for the `send_action_req` function. If specified, a confirmed action request operation is issued and the callback function is invoked when the response is received. If not specified, an unconfirmed action request operation is issued and no response is generated.	Optional
`MessId id`	Provides a unique identifier for a particular action request operation. If specified, the value for this parameter is generated and set by the `send_action_req` function.	Optional
`const MessScope scope`	Defines the type of scoping used for this request operation. If not specified, it defaults to BASE_OBJECT_ONLY.	Optional
`const Asn1Value filter`	Defines a filter to be passed by all objects selected by the scoping parameter. If not specified, it defaults to a filter that matches all managed objects selected by the scope parameter.	Optional
`const MessSync sync`	Defines the type of synchronization used for this request operation. If not specified, it defaults to BEST_EFFORT.	Optional
`const Asn1Value access`	Reserved parameter that should not be used at this time.	Not available

6.4.5.3 `send_action_req` Examples

Code Example 6-10 `send_action_req` Example

```
Result
send_registerLocal_action(char *local_sys, char *remote_sys)
{
    Asn1Value actual_oc;
    Asn1Value dalarm_fdn;
    Asn1Value em_mis_rdn;
    Asn1Value dalarm_rdn;
    Asn1Value act_type;
    Asn1Value act_info;
    char      local_sys_id[100];
```

Code Example 6-10 `send_action_req` Example

```
Result
    char      remote_sys_id[100];
    Asn1Value local_id;
    Asn1Value remote_id;
    Asn1Value log_id;

    //********************************************
    // Confirmed Action request
    //********************************************
    // Send a confirmed registerLocal action request to an
    // instance of the distributed alarm manager object class.
    // The distributed alarm log manager that the request
    // is sent to is contained under the system identified
    // by the remote_sys parameter. The local_sys parameter
    // is the name of the system that contains the distributed
    // alarm log manager issuing this request.

    VTRY {

    // Encode oc OID. In this example, actualClass is used
    // instead of the OID for the log managed object
    // class.
    // Note: CMIP requires a TAG_CONT(0) encoding for the
    // object class rather than TAG_OID (refer to x711.asn1
    // for encoding spec).

        TTRYRES(actual_oc.encode_oid(TAG_CONT(0),
            Oid("2.9.3.4.3.42")));

    // Encode the distinguished name for an instance of the
    // distributed alarm log manager managed object class. The
    // distributed alarm log manager is always named ???
    // Either the distinguished name form (TAG_CONT(2))
    // or the local distinguished name form (TAG_CONT(4)) can be
    // used with the action operation. The distinguished name form
    // is used in this example.
    // The get_sys_dn and get_graphstr_rdn functions are included
    // in Section 6.5.

        TTRYRES(get_sys_dn(remote_sys, dalarm_fdn));
        TTRYRES(get_graphstr_rdn("2.9.3.5.7.11",
            "EM-MIS", em_mis_rdn));
        TTRYRES(get_graphstr_rdn(
```

Code Example 6-10 `send_action_req` Example

```
Result
           Oid("1.3.6.1.4.1.42.2.2.2.300.7.1", "Distrib-
AlarmLog",
               dalarm_rdn));
       TTRYRES(dalarm_fdn.add_component(em_mis_rdn));
       TTRYRES(dalarm_fdn.add_component(dalarm_rdn));

    // Encode the action type and action info parameters
    // for the action operation. The ASN.1 for the registerLocal
    // action_info is:
    // LocalRegistrar ::= SEQUENCE
    // {
    //     receiverMIS        SystemId,
    //     senderMIS          SystemId,
    //     logId              SimpleNameType
    // }
    // The SystemIds for each of the systems also include
    // a port number -- 5555 in this example. Port 5555
    // is the default CMIP/LPP port used by the MIS.
    // The CMIP spec specifies TAG_CONT(2) as the Tag for
    // the action type.
       TTRYRES(act_type.encode_oid(TAG_CONT(2),
           Oid("1.3.6.1.4.1.42.2.2.2.300.9.4")));

       sprintf(local_sys_id, "%s:%s", local_sys, "5555");
       TTRYRES(local_id.encode_octets(TAG_GRAPHSTR,
           DataUnit((char *)&local_sys_id)));
       sprintf(remote_sys_id,"%s:%s", remote_sys, "5555");
       TTRYRES(remote_id.encode_octets(TAG_GRAPHSTR,
           DataUnit((char *)&remote_sys_id)));
       TTRYRES(log_id.encode_octets(TAG_GRAPHSTR, "AlarmLog"));

       TTRYRES(act_info.start_construct(TAG_SEQ));
       TTRYRES(act_info.add_component(remote_id));
       TTRYRES(act_info.add_component(local_id));
       TTRYRES(act_info.add_component(log_id));

    // Send the confirmed action request. No user data is
    // specified for the callback parameter. The action_req_cb
    // function is included as part of the services
    // API examples.

       oamsvc_test.print("About to issue Confirmed Action
               Request\n");
```

Code Example 6-10 `send_action_req` Example

```
Result
        if (send_action_req(actual_oc, dalarm_fdn, act_type,
            act_info, Callback(action_req_cb, 0)) != OK)
             objsvc_test.print("Error issuing Confirmed Action
                Request\n");
            return NOT_OK;
        }
        else
            objsvc_test.print("Issued Confirmed Action
Request\n");
    }
    VBEGHANDLERS
    VCATCHALL {
        objsvc_test.print("\nError encoding Action Request\n");
        return NOT_OK;
    }
    VENDHANDLERS
    return OK;
}
```

6.4.6 Action Response Callback

This callback function is used only in conjunction with confirmed action requests. The name of the callback function must match the name of the callback function specified in the callback parameter of the `send_action_req` function.

6.4.6.1 *Interface Signature*

```
void  user-provided-action-response-callback(
    Ptr userdata,       // Pointer to user supplied data
    Ptr message);       // Pointer to ActionRes message
```

6.4.6.2 *Action Response Callback Parameter Description*

Table 6-7 describes the action response callback parameters.

Table 6-7 Action Response Callback Parameter Table

Parameter	**Description**
`Ptr userdata`	A `void *` pointer to data specified by the application developer in the callback parameter of the `send_action_req` function.
`Ptr message`	A `void *` pointer to the ActionRes message generated in response to the `send_action_req` function.

6.4.6.3 *Action Response Callback Example*

The header file `dalarm.hh` is needed to successfully compile and link this example. The header file is needed to obtain the type definition for the `RegState` data type.

Code Example 6-11 Action Response Callback Example

```
void
action_req_cb(Ptr, Ptr action_response_msg)
{
    objsvc_test.print("Action Request callback\n");
    Message *resp = (Message *)action_response_msg;

    VTRY {
        if ( resp->type() == Action_RES)  {
            ActionRes *a_resp = (ActionRes *)resp;
            a_resp->print(objsvc_test);

        // Decode the action reply field
        // LocalRegistrarReply ::= SEQUENCE
        // {
        //      senderMIS          SystemId,
```

Code Example 6-11 Action Response Callback Example

```
void
        //      logId               SimpleNameType,
        //      regStatus           RegistrationState
        // }

            Asn1Value   remote_id, log_id, reg_state;
            DataUnit    sndr_du, log_du;
            I32         reg_val;
            RegState    reg;

            TTRYRES(a_resp->action_reply.
                first_component(remote_id));
            TTRYRES(remote_id.decode_octets(sndr_du));
            TTRYRES(a_resp->action_reply.next_component(
                remote_id, log_id));
            TTRYRES(log.decode_octets(log_du));
            TTRYRES(a_resp->action_reply.next_component(
                log_id, reg_state));
            TTRYRES(reg_state.decode_enum(reg_val));
            reg = (RegState)reg_val;

        // Now do any other action response processing.
        //  ...

        }
        else if ( resp->is_error() ) {
            objsvc_test.print("Action response error
received\n");
            objsvc_test.print( "message type = %s\n",
                MessType_fmt(resp->type()));
            resp->print(objsvc_error);
        }
        else {
            objsvc_test.print(
                "Unexpected or invalid response received\n");
            objsvc_test.print( "message type = %s\n",
                MessType_fmt(resp->type()));
            resp->print(objsvc_error);
        }
    }
    VBEGHANDLERS
    VCATCHALL {
        oamsvc_test.print("Error processing response for
Action\n");
```

Code Example 6-11 Action Response Callback Example

```
void
    }
    VENDHANDLERS

    if ( resp )
        Message:: delete_message(resp);
}
```

6.4.7 *Create Request Service*

6.4.7.1 *Interface Signature*

```
Result send_create_req(

    const Asn1Value &oc,

    const Asn1Value &oi,

    const Asn1Value attr_value_list = Asn1Value(),

    const Callback cb = 0,        // Default: Unconfirmed

    MessId *id = 0,

    const Asn1Value superior_oi = Asn1Value(),

    const Asn1Value reference_oi = Asn1Value(),

    const Asn1Value access = Asn1Value()); / Default: No access
ctrl
```

6.4.7.2 `send_create_req` *Parameter Descriptions*

The following table describes the `send_create_req` parameters:

Table 6-8 describes the `send_create_req` parameters.

Table 6-8 `send_create_req` Parameter Table

Parameter	**Description**	**Required/ Optional**
`const Asn1Value oc`	OID that specifies the class of the managed object.	Required
`const Asn1Value oi`	Distinguished name or local distinguished name for the managed object.	Required
`const Asn1Value attr_value_list`	Set of attribute ID and attribute value pairs that specify the attributes assigned new values by a `send_create_req` service request operation. The values specified in the `send_create_req` service function override the corresponding attributes from the reference object (if specified) or from the default value specified in the GDMO definition for the managed object class. Although this is an optional parameter, attribute values must be specified for all mandatory attributes defined for a GDMO managed object class, specified by a default value, by a reference object, or by this parameter.	Optional
`const Callback cb`	Specifies the name of a callback function invoked when a response to the create request operation is received. Can be used to specify user data passed to the callback function when it is invoked. Callback parameter is optional for the `send_create_req` function. If specified, a confirmed create request operation is issued and the callback function is invoked when the response is received. If not specified, an unconfirmed create request operation is issued and no response is generated.	Optional
`MessId id`	Provides a unique identifier for a particular create request operation. If specified, the value for this parameter is generated and set by the `send_create_req` function.	Optional
`const Asn1Value superior_oi`	Distinguished name or local distinguished name for an existing superior object under which a managed object is to be created. This optional parameter should not be specified if a value is specified for the `oi` parameter. The managed object created is contained under the superior object in the MIT.	Optional
`const Asn1Value reference_oi`	Distinguished name or local distinguished name for a reference object. If this parameter is specified, it must specify the name of a managed object of the same class as the managed object to be created. The attribute values of the reference object become default values for the new managed object for any attributes not specified in the value for the `attr_value_list` parameter.	Optional
`const Asn1Value access`	Reserved parameter, not to be used at this time.	Not available

6.4.7.3 `send_create_req` Examples

The `mysystem.gdmo` and `mysystem.asn1` files must be loaded into the MDR for this example to work properly. The `mysystem` files can be found in `/opt/SUNWconn/em/src/scenario/example1`. The `README` file in this directory also describes how to load the `mysystem` files.

Code Example 6-12 `send_create_req` Example

```
Result
create_mySystem_object(char *system_name, char *mySystem_name)
{
    Asn1Value mySystem_oc;
    Asn1Value mySystem_fdn;
    Asn1Value mySystem_rdn;

    //******************************************
    // Confirmed create request
    //******************************************
    // Send a confirmed create request for an
    // instance of the mySystem manager object class.
    // Instances of the mySystem class are contained by
    // instances of the System class. The system_name
    // parameter specifies the name of the system to contain
    // the new instance. The mySystem_name parameter is used
    // to specify the name of the instance to be created.

    VTRY {

    // Encode oc OID. In this example, the OID for the
    // mySystem class is used.
    // Note: CMIP requires a TAG_CONT(0) encoding for the
    // object class rather than TAG_OID (refer to x711.asn1
    // for encoding spec).

        TTRYRES(mySystem_oc.encode_oid(
            TAG_CONT(0),Oid("1.2.3.4.5.6.3.10")));

    // Encode the distinguished name for an instance of the
    // mySystem managed object class. Instances of the mySystem
    // class are named using the systemId attribute in the class.
    // Either the distinguished name form (TAG_CONT(2))
    // or the local distinguished name form (TAG_CONT(4)) can be
    // used with the create operation. The distinguished name form
    // is used in this example.
```

Code Example 6-12 `send_create_req` Example

```
Result
    // The get_sys_dn and get_graphstr_rdn functions are included
    // in Section 6.5. The definition for the sys_id (systemId)
OID
    // is included in Section 6.5.

        TTRYRES(get_sys_dn(system_name, mySystem_fdn));
        TTRYRES(get_graphstr_rdn(sys_id, // systemId OID
            mySystem_name, mySystem_rdn));
        TTRYRES(mySystem_fdn.add_component(mySystem_rdn));

    // Encode the attribute list for the create request
    // All mandatory attributes must be specified as part
    // of the create request (unless the GDMO defines a
    // initial value sub-clause for the attribute, or
    // specifies a reference object that contains the
    // attribute).
    // Note: CMIP requires a TAG_CONT(7) encoding for the
    // attribute list rather than TAG_SEQ (refer to x711.asn1
    // for encoding spec).

        Asn1Value attrlist;

        Asn1Value opState;
        Asn1Value usState;
        Asn1Value sysTitle;
        Asn1Value wInt;
        Asn1Value rString;

        Asn1Value sysIdO;
        Asn1Value opStateO;
        Asn1Value usStateO;
        Asn1Value sysTitleO;
        Asn1Value wIntO;
        Asn1Value rStringO;

        Asn1Value opStateV;
        Asn1Value usStateV;
        Asn1Value sysTitleV;
        Asn1Value wIntV;
        Asn1Value rStringV;

        // Initialize the OIDs.
```

Code Example 6-12 `send_create_req` Example

```
Result
        Oid sysIdOid("2.9.3.2.7.4"); //    systemId
        Oid opStateOid("2.9.3.2.7.35"); // operationalState
        Oid usStateOid("2.9.3.2.7.39"); // usageState
        Oid sysTitleOid("2.9.3.2.7.5"); // systemTitle
        Oid wIntOid("1.2.3.4.5.6.7.10"); // writeableInteger
        Oid rStringOid("1.2.3.4.5.6.7.11"); // readableString

        // Initialize the five mandatory attributes
        // in the mySystem class.

        TTRYRES(opStateO.encode_oid(TAG_CONT(0), opStateOid));
        TTRYRES(opStateV.encode_enum(TAG_ENUM, 1)); // enabled
        TTRYRES(opState.start_construct(TAG_SEQ));
        TTRYRES(opState.add_component(opStateO));
        TTRYRES(opState.add_component(opStateV));

        TTRYRES(usStateO.encode_oid(TAG_CONT(0), usStateOid));
        TTRYRES(usStateV.encode_enum(TAG_ENUM, 1)); // active
        TTRYRES(usState.start_construct(TAG_SEQ));
        TTRYRES(usState.add_component(usStateO));
        TTRYRES(usState.add_component(usStateV));

        TTRYRES(sysTitleO.encode_oid(TAG_CONT(0), sysTitleOid));
        TTRYRES(sysTitleV.encode_null(TAG_NULL));
        TTRYRES(sysTitle.start_construct(TAG_SEQ));
        TTRYRES(sysTitle.add_component(sysTitleO));
        TTRYRES(sysTitle.add_component(sysTitleV));

        TTRYRES(wIntO.encode_oid(TAG_CONT(0), wIntOid));
        TTRYRES(wIntV.encode_int(TAG_INT, 5));
        TTRYRES(wInt.start_construct(TAG_SEQ));
        TTRYRES(wInt.add_component(wIntO));
        TTRYRES(wInt.add_component(wIntV));

        TTRYRES(rStringO.encode_oid(TAG_CONT(0), rStringOid));
        TTRYRES(rStringV.encode_octets(TAG_GRAPHSTR,DataUnit
            ("test1")));
        TTRYRES(rString.start_construct(TAG_SEQ));
        TTRYRES(rString.add_component(rStringO));
        TTRYRES(rString.add_component(rStringV));

        // Create the attribute list.
```

Code Example 6-12 `send_create_req` Example

```
Result

TTRYRES(attrlist.start_construct(TAG_CONT(7)));//Implicit
            set of
        TTRYRES(attrlist.add_component(opState));
        TTRYRES(attrlist.add_component(usState));
        TTRYRES(attrlist.add_component(sysTitle));
        TTRYRES(attrlist.add_component(wInt));
        TTRYRES(attrlist.add_component(rString));

    // Send the confirmed create request. No user data is
    // specified for the callback parameter. The create_req_cb
    // function is included as part of the services
    // API examples.

        oamsvc_test.print("About to issue Confirmed Create
            Request\n");
        if (send_action_req(mySystem_oc, mySystem_fdn, attrlist,
            Callback(create_req_cb, 0)) != OK)
             objsvc_test.print("Error issuing Confirmed Create
                Request\n");
            return NOT_OK;
        }
        else
            objsvc_test.print("Issued Confirmed Create
Request\n");
    }
    VBEGHANDLERS
    VCATCHALL {
        objsvc_test.print("\nError encoding Create Request\n");
        return NOT_OK;
    }
    VENDHANDLERS
    return OK;
}
```

6.4.8 Create Response Callback

This function is used only in conjunction with confirmed create requests. The name of the callback function must match the name of the callback function specified in the callback parameter of the `send_create_req` function.

6.4.8.1 Interface Signature

```
void user-provided-create-response-callback(
    Ptr userdata,                    // Pointer to user supplied
data
    Ptr message);                    // Pointer to CreateRes message
```

6.4.8.2 Create Response Callback Parameter Descriptions

Table 6-9 describes these parameters.

Table 6-9 Create Response Callback Parameter Table

Parameter	**Description**
`Ptr userdata`	A `void *` pointer to data specified by the application developer in the callback parameter of the `send_create_req` function.
`Ptr message`	A `void *` pointer to the `CreateRes` message generated in response to the `send_create_req` function.

6.4.8.3 Create Response Callback Example

The `mysystem.gdmo` and `mysystem.asn1` files must be loaded into the MDR for this example to work properly. The `mysystem` files can be found in `/opt/SUNWconn/em/src/scenario/example1`. The `README` file in this directory also describes how to load the `mysystem` files.

Code Example 6-13 Create Response Callback Example

```
void
create_req_cb(Ptr, Ptr create_response_msg)
{
    objsvc_test.print("Create Request callback\n");
    Message *resp = (Message *)create_response_msg;

    VTRY {
        if ( resp->type() == CREATE_RES)  {
            CreateRes *cr_resp = (CreateRes *)resp;
            cr_resp->print(objsvc_test);

        // The Asn1Value decoding functions, including the
        // member functions get_first_component and
```

Code Example 6-13 Create Response Callback Example

```
void
        // get_next_component can be used to examine the member
data
        // included in the response message at this point

        //  ...

        }
        else if ( resp->is_error() ) {
            objsvc_test.print("Action response error
received\n");
            objsvc_test.print( "message type = %s\n",
                MessType_fmt(resp->type()));
            resp->print(objsvc_error);
        }
        else {
            objsvc_test.print(
                "Unexpected or invalid response received\n");
            objsvc_test.print( "message type = %s\n",
                MessType_fmt(resp->type()));
            resp->print(objsvc_error);
        }
    }
    VBEGHANDLERS
    VCATCHALL {
        oamsvc_test.print("Error processing response for
Action\n");
    }
    VENDHANDLERS

    if ( resp )
        Message:: delete_message(resp);
}
```

6.4.9 Delete Request Service

This section includes the interface signature, send_delete_req parameters, and an example.

6.4.9.1 *Interface Signature*

Code Example 6-14 Delete Request Service, Interface Signature

```
Result send_delete_req(
    const Asn1Value &oc,
    const Asn1Value &oi,
    const Callback cb = 0,            // Default: Unconfirmed
    MessId *id = 0,
    const MessScope scope = MessScope(), // Default: Base object
only
    const Asn1Value filter = Asn1Value(), // Default: Matches all
    const MessSync sync = BEST_EFFORT,
    const Asn1Value access = Asn1Value()); // Default: No access
cntrl
```

6.4.9.2 `send_delete_req` *Parameter Descriptions*

Table 6-10 describes the `send_delete_req` parameters.

Table 6-10 `send_delete_req` Parameter Table

Parameter	**Description**	**Required/ Optional**
`const Asn1Value oc`	OID that specifies the class of the managed object.	Required
`const Asn1Value oi`	Distinguished name or local distinguished name for the managed object.	Required
`const Callback cb`	Specifies the name of a callback function invoked when a response to the delete request operation is received. Can also be used to specify user data passed to the callback function when it is invoked. The callback parameter is optional for the `send_delete_req` function. If specified, a confirmed delete request operation is issued and the callback function is invoked when the response is received. If not specified, an unconfirmed delete request operation is issued and no response is generated.	Optional
`MessId id`	Provides a unique identifier for a particular delete request operation. If specified, the value for this parameter is generated and set by the `send_delete_req` function.	Optional
`const MessScope scope`	Defines the type of scoping used for this request operation. If not specified, it defaults to `BASE_OBJECT_ONLY`.	Optional

Table 6-10 send_delete_req Parameter Table

Parameter	Description	Required/ Optional
`const Asn1Value filter`	Defines a filter to be passed by all objects selected by the scoping parameter. If not specified, it defaults to a filter that matches all managed objects selected by the scope parameter.	Optional
`const MessSync sync`	Defines the type of synchronization used for this request operation. If not specified, it defaults to BEST_EFFORT.	Optional
`const Asn1Value access`	Reserved parameter, not to be used at this time.	Not available

6.4.9.3 send_delete_req Examples

The `mysystem.gdmo` and `mysystem.asn1` files must be loaded into the MDR for this example to work properly. The `mysystem` files can be found in `/opt/SUNWconn/em/src/scenario/example1`. The `README` file in this directory also describes how to load the `mysystem` files.

Delete Request: Base Object Only

Code Example 6-15 Delete Request, Base Object Example

```
Result
delete_mySystem_object(char *system_name, char *mySystem_name)
{
    Asn1Value mySystem_oc;
    Asn1Value mySystem_fdn;
    Asn1Value mySystem_rdn;

    //*******************************************
    // Confirmed delete request
    //*******************************************
    // Send a delete request to an instance of the
    // mySystem managed object class. The mySystem
    // instance to be deleted is contained by the system
    // object specified by system_name. The name of the
    // instance of the mySystem class to delete is specified
    // by the mySystem_name parameter.

    VTRY {
```

Code Example 6-15 Delete Request, Base Object Example

```
Result

    // Encode oc OID. In this example, the OID for the
    // mySystem class is used.
    // Note: CMIP requires a TAG_CONT(0) encoding for the
    // object class rather than TAG_OID (refer to x711.asn1
    // for encoding spec).

        TTRYRES(mySystem_oc.encode_oid(
            TAG_CONT(0),Oid("1.2.3.4.5.6.3.10")));

        // Encode the distinguished name for an instance of the
    // mySystem managed object class. Instances of the mySystem
    // class are named using the systemId attribute in the class.
    // Either the distinguished name form (TAG_CONT(2))
    // or the local distinguished name form (TAG_CONT(4)) can be
    // used with the action operation. The distinguished name form
    // is used in this example.
    // The get_sys_dn and get_graphstr_rdn functions are included
    // in Section 6.5. The definition for the sys_id (systemId)
OID
    // is included in Section 6.5.

        TTRYRES(get_sys_dn(system_name, mySystem_fdn));
        TTRYRES(get_graphstr_rdn(sys_id, // systemId OID
            mySystem_name, mySystem_rdn));
        TTRYRES(mySystem_fdn.add_component(mySystem_rdn));

    // Send the confirmed Delete Request. No user data is
    // specified for the callback parameter. The delete_req_cb
    // function is included as part of the Services API examples.

        objsvc_test.print("About to issue confirmed Delete
            Request\n");
        if (send_delete_req(mySystem_oc, mySystem_fdn,
            Callback(delete_req_cb, 0)) != OK) {
            objsvc_test.print("Error issuing confirmed Delete
Request\n");
            return NOT_OK;
        }
        else
            objsvc_test.print("Issued confirmed Delete
Request\n");
    }
```

Code Example 6-15 Delete Request, Base Object Example

```
Result
    VBEGHANDLERS
    VCATCHALL {
        objsvc_test.print("\nError encoding Delete Request\n");
        return NOT_OK;
    }
    VENDHANDLERS
    return OK;
}
```

Delete Request: Scoped Operation

Code Example 6-16 Delete Request, Scoped Operation

```
Result
delete_mySystem_object(char *system_name)
{
    Asn1Value actual_oc;
    Asn1Value system_fdn;

    //*******************************************
    // Confirmed delete request
    //*******************************************
    // Send a delete request to delete all instances
    // of the mySystem managed object class below a
    // system object. The system "base" object is
    // specified by the system_name parameter.
    // A CMIS filter is used to select only instances
    // of the mySystem class.

    VTRY {

    // Encode oc OID. In this example, actualClass is used
    // instead of the OID for the log managed object class.
    // Note: CMIP requires a TAG_CONT(0) encoding for the
    // object class rather than TAG_OID (refer to x711.asn1
    // for encoding spec).

        TTRYRES(actual_oc.encode_oid(TAG_CONT(0),
            Oid("2.9.3.4.3.42")));

    // Encode the distinguished name for the instance of the
    // system managed object class that is the base object for
    // the scoped operation.
```

Code Example 6-16 Delete Request, Scoped Operation

```
Result
    // Either the distinguished name form (TAG_CONT(2))
    // or the local distinguished name form (TAG_CONT(4)) can be
    // used with the delete operation. The distinguished name form
    // is used in this example.
    // The get_sys_dn and get_graphstr_rdn functions are included
    // in Section 6.5. The definition for the sys_id (systemId)
OID
    // is included in Section 6.5.

        TTRYRES(get_sys_dn(system_name, system_fdn));

    // Encode the CMIS Filter. This filter checks for
    // "managedObjectClass == mySystem"

        Asn1Value cmis_filter;
        Asn1Value filter_item;
        Asn1Value mOC_OID;
        Asn1Value mySystem_oc_OID;

        TTRYRES(cmis_filter.start_construct(TAG_CONT(8));
        TTRYRES(filter_item.start_construct(TAG_CONT(0));
        TTRYRES(mOC_OID.encode_oid(               // ISO DMI
            TAG_CONT(0),Oid("2.9.3.2.7.60")));    //
managedObjectClass
        TTRYRES(mySystem_oc_OID.encode_oid(
            TAG_CONT(0),Oid("1.2.3.4.5.6.3.10")));
        TTRYRES(filter_item.add_component(mOC_OID));
        TTRYRES(filter_item.add_component(mySystem_oc_OID));
        TTRYRES(cmis_filter.add_component(filter_item));

    // Send the confirmed scoped Delete Request. No user data is
    // specified for the callback parameter. The delete_req_cb
    // function is included as part of the Services API examples.

        objsvc_test.print(
            "About to issue confirmed scoped Delete Request\n");
        if (send_delete_req(mySystem_oc, mySystem_fdn,
             Callback(delete_req_cb, 0),
            MessScope(NTH_LEVEL, 1), cmis_filter) != OK) {
             objsvc_test.print(
                "Error issuing confirmed scoped Delete
Request\n");
            return NOT_OK;
```

Code Example 6-16 Delete Request, Scoped Operation

```
Result
        }
        else
            objsvc_test.print("Issued confirmed scoped Delete
                Request\n");
    }
    VBEGHANDLERS
    VCATCHALL {
        objsvc_test.print("\nError encoding scoped Delete
                Request\n");
        return NOT_OK;
    }
    VENDHANDLERS
    return OK;
}
```

6.4.10 *Delete Response Callback*

This callback function is used only in conjunction with the confirmed delete request service. The name of the callback function must match the name of the callback function specified in the callback parameter of the `send_delete_req` function.

6.4.10.1 *Interface Signature*

```
void user-provided-delete-response-callback(
    Ptr userdata,              // Pointer to user supplied data
    Ptr message);              // Pointer to DeleteRes message
```

6.4.11 *Delete Response Callback Parameter Description*

Table 6-11 describes the delete response callback parameters.

Table 6-11 Delete Response Callback Parameter Table

Parameter	**Description**
`Ptr userdata`	A `void *` pointer to data specified by the application developer in the callback parameter of the `send_delete_req` function.

Table 6-11 Delete Response Callback Parameter Table

Parameter	**Description**
`Ptr message`	A `void *` pointer to the `DeleteRes` message generated in response to the `send_delete_req` function.

6.4.11.1 Delete Response Callback Example

The `mysystem.gdmo` and `mysystem.asn1` files must be loaded into the MDR for this example to work properly. The `mysystem` files can be found in `/opt/SUNWconn/em/src/scenario/example1`. The `README` file in this directory also describes how to load the `mysystem` files.

Delete Response Callback Function: Single Response (Base Object Only)

Code Example 6-17 Delete Response Callback Function

```
void
get_req_cb(Ptr, Ptr delete_response_msg)
{
    objsvc_test.print("Get Request callback\n");
    Message *resp = (Message *)delete_response_msg;

    VTRY {
        if ( resp->type() == DELETE_RES)  {
            DeleteRes *d_resp = (DeleteRes *)resp;
            d_resp->print(objsvc_test);

        // The Asn1Value decoding functions, including the
        // member functions get_first_component and
        // get_next_component can be used to examine the member
data
        // included in the response message at this point

        //  ...

        }
        else if ( resp->is_error() ) {
            objsvc_test.print("Delete response error
received\n");
            objsvc_test.print( "message type = %s\n",
                MessType_fmt(resp->type()));
            resp->print(objsvc_error);
        }
        else {
            objsvc_test.print(
                "Unexpected or invalid response received\n");
            objsvc_test.print( "message type = %s\n",
                MessType_fmt(resp->type()));
            resp->print(objsvc_error);
        }
    }
    VBEGHANDLERS
    VCATCHALL {
        objsvc_test.print("Error processing response for
Delete\n");
        resp->print(objsvc_error);
    }
    VENDHANDLERS
```

Code Example 6-17 Delete Response Callback Function

```
void

    if ( resp )
        Message:: delete_message(resp);
}
```

Delete Response Callback Function: Multiple Responses (Scoped Operation)

Code Example 6-18 Delete Response Callback Function

```
static int d_resp_count = 0;
static int d_err_count = 0;
static int d_unknown_count = 0;
void
sg_req_cb(Ptr userdata, Ptr delete_response_msg)
{
    objsvc_test.print("Scoped Delete Request callback:");
    Message *resp = (Message *)delete_response_msg;

    VTRY {
        if ( resp->type() == DELETE_RES)  {
            DeleteRes *sd_resp = (DeleteRes *)resp;
            if (sd_resp->linked) {
                objsvc_test.print("*** LINKED Response ***\n");
                resp_count++;
                sd_resp->print(objsvc_test);

            // The Asn1Value decoding functions, including the
            // member functions get_first_component and
            // get_next_component can be used to examine the
member
            // data included in the response message at this point

            //  ...

            } else {
                objsvc_test.print("**** Final Response ****\n");
                sd_resp->print(objsvc_test);
                objsvc_test.print("Valid: %d, Error: %d, Invalid:
                    %d\n",
                resp_count, err_count, unknown_count);
```

Code Example 6-18 Delete Response Callback Function

```
static int d_resp_count = 0;
               // Final response processing can be performed here.
The
               // final response message does not contain any
attribute
               // value data.

               //  ...

               }
          }
          else if ( resp->is_error() ) {
               objsvc_test.print("Error Response\n");
               objsvc_test.print( "message type = %s\n",
                    MessType_fmt(resp->type()));
               resp->print(objsvc_error);
               err_count++;
          }
          else {
               objsvc_test.print("Invalid Message\n");
               objsvc_test.print( "message type = %s\n",
                    MessType_fmt(resp->type()));
               resp->print(objsvc_error);
               unknown_count++;
          }
     }
     VBEGHANDLERS
     VCATCHALL {
          objsvc_test.print(
               "\nError processing response for Scoped Delete\n");
     }
     VENDHANDLERS

     if ( resp )
          Message:: delete_message(resp);
}
```

6.4.12 Event Report Request Service (Unconfirmed)

This section includes the interface signature, `send_event_req` parameters, and an example.

6.4.12.1 *Interface Signature*

Code Example 6-19 Event Report Request, Interface Signature

```
Result send_event_req(
    const Asn1Value &oc,
    const Asn1Value &oi,
    const Asn1Value &event_type,
    const Asn1Value event_info = Asn1Value(), // Default: No event
info
    const Asn1Value event_time = Asn1Value(), // Default: No value
    const Callback cb = 0,            // Default: Unconfirmed
    MessId *id = 0);
```

Note – The `send_event_req` service supports only unconfirmed operations. Specifying a callback generates a runtime error.

6.4.12.2 *`send_event_req` Parameter Descriptions*

Table 6-12 describes the `send_event_req` parameters.

Table 6-12 `send_event_req` Parameter Table

Parameter	**Description**	**Required/ Optional**
`const Asn1Value oc`	OID that specifies the class of the base managed object.	Required
`const Asn1Value oi`	Distinguished name or local distinguished name for the base managed object.	Required
`const Asn1Value event_type`	OID that specifies the type of notification generated by the `send_event_req` function.	Required
`const Asn1Value event_info`	Specifies any event information associated with the type of notification generated. This is an optional parameter but must be present if a `WITH INFORMATION SYNTAX` construct is specified as part of the GDMO definition for the notification type.	Optional
`const Asn1Value event_time`	This is the time at which the notification is generated.	Optional

Table 6-12 `send_event_req` Parameter Table

Parameter	Description	Required/ Optional
`const Callback cb`	This is an optional reserved parameter for the event report request operation and should not be specified. Only unconfirmed (no callback function) event request operations are currently supported using this function call. If a value for this parameter is specified (in order to specify an `id` parameter, the `cb` parameter needs to be specified), it must either be set to `Callback()` or `Callback(0,0)`. Any other value generates an error when the `send_event_req` function is invoked.	Optional reserved
`MessId id`	Message identifier that uniquely identifies this request operation. If specified, the value for this parameter is generated and set by the `send_event_req` function.	Optional

6.4.12.3 `send_event_req` Example

The GDMO and definition for the notification generated by this example is as follows:

Code Example 6-20 `send_event_req` Example

```
connectivityChange NOTIFICATION
    BEHAVIOUR connectivityChangeBehaviour BEHAVIOUR DEFINED AS
        !Generated by an instance of the connectMgr object
        when the state of a connection between two MISs
        changes.!;
    ;
    WITH INFORMATION SYNTAX ConnectMgr-
ASN1.ConnectivityChangeInfo
    AND ATTRIBUTE IDS
        ConnectivityChangeDefinition
connectivityChangeDefinition;
REGISTERED AS { connectivityMgmt 10 1 };
The corresponding ASN.1 definitions are as follows:
ConnectMgr-ASN1 { 6 2 3 4 5 6 7 2 1 }
            ...
       ConnectState ::= ENUMERATED
{
        NotConnected(0),
        Connected(1),
        ErrorDisconnect(2),
        ResyncDisconnect(3)
}
```

Code Example 6-20 `send_event_req` Example

```
connectivityChange NOTIFICATION
           ...
ConnectivityChangeDefinition ::= SEQUENCE
{
    RemoteMIS           SystemId,
    PreviousState       ConnectState,
    CurrentState        ConnectState
}
ConnectivityChangeInfo ::= ConnectivityChangeDefinition
Example code to generate notification using send_event_req
function is:
Result
send_conn_change_event(char *from_sys, char *remote_sys)
{
    Asn1Value conn_oc;
    Asn1Value conn_oi;
    Asn1Value em_mis_rdn;
    Asn1Value conn_rdn;
    Asn1Value conn_evt_type;
    Asn1Value conn_evt_info;
    Asn1Value rmt_id;
    Asn1Value rmt_prev;
    Asn1Value rmt_curr;

    //*******************************************
    // Unconfirmed event report request
    //*******************************************
    // Issue a connectivity change notification
    // This notification is defined by the connection
    // manager object (refer to connmgr.gdmo/connmgr.asn1)
    // and is issued when the connectivity state between
    // two MISs changes. In this example, the two MISs
    // are from_sys (which generates the notification) and
    // remote_sys.

    VTRY {

    // Encode oc OID for connection manager object class
    // which is defined in connmgr.gdmo. Note: CMIP requires
    // a TAG_CONT(0) encoding for the object class rather
    // than TAG_OID (refer to x711.asn1 for encoding spec).

        TTRYRES(conn_oc.encode_oid(
            TAG_CONT(0),Oid("1.3.6.1.4.1.42.2.2.2.201.3.1")));
```

Code Example 6-20 `send_event_req` Example

```
connectivityChange NOTIFICATION

    // Encode the distinguished name for the connection manager
    // managed object that generates the notification.
    // The send_event_req function requires the distinguished
    // name form (TAG_CONT(2)) for the object instance.

        TTRYRES(get_sys_dn(from_sys, conn_oi));
        TTRYRES(get_graphstr_rdn(
            "2.9.3.5.7.11","EM-MIS", em_mis_rdn));
        TTRYRES(get_graphstr_rdn(
            "1.3.6.1.4.1.42.2.2.2.201.7.3","ConnectMgr",
            conn_rdn));
        TTRYRES(conn_oi.add_component(em_mis_rdn));
        TTRYRES(conn_oi.add_component(conn_rdn));

    // Encode the connectivityChange event type and event info
    // The connectivityChange OID is defined in connmgr.gdmo.
    // The format of the corresponding event info is defined
    // in connmgr.asn1.
    // Note: CMIP requires a TAG_CONT(6) for the event type OID
    // rather than a TAG_OID and also requires a TAG_CONT(8)
    // encoding for the event info rather than a TAG_SEQ.
        TTRYRES(conn_evt_type.encode_oid(
            TAG_CONT(6),Oid("1.3.6.1.4.1.42.2.2.2.201.10.1")));
        TTRYRES(conn_evt_info.start_construct(TAG_CONT(8)));
        TTRYRES(rmt_id.encode_octets(
            TAG_GRAPHSTR, DataUnit(remote_sys)));
        TTRYRES(rmt_prev.encode_enum(TAG_ENUM, 0)); //
notConnected
        TTRYRES(rmt_curr.encode_enum(TAG_ENUM, 1)); // connected
        TTRYRES(conn_evt_info.add_component(rmt_id));
        TTRYRES(conn_evt_info.add_component(rmt_prev));
        TTRYRES(conn_evt_info.add_component(rmt_curr));

    // Send the Event Report Request

        objsvc_test.print(
            "About to issue Unconfirmed EVENT REPORT Request\n");
        if (send_event_req(conn_oc, conn_oi, conn_evt_type,
            conn_evt_info) != OK) {
            objsvc_test.print(
                "Error issuing Unconfirmed EVENT REPORT
Request\n");
```

Code Example 6-20 `send_event_req` Example

```
connectivityChange NOTIFICATION
            return NOT_OK;
        }
        else
            objsvc_test.print("Issued Unconfirmed EVENT REPORT
                Request\n");
    }
    VBEGHANDLERS
    VCATCHALL {
        objsvc_test.print("\nError encoding EVENT REPORT
            Request\n");
        return NOT_OK;
    }
    VENDHANDLERS
    return OK;
}
```

6.4.13 Event Report Response Callback

This callback function is not supported for the EM services interface.

6.5 Supporting Functions for Example Code

This section includes the following information:

- *Debugging Flags*
- *get_sys_dn Function*
- *get_graphstr_rdn Functions*

6.5.1 Debugging Flags

The following code lines must be included in the file that contains the example functions.

```
    Debug_on(objsvc_test)
    Debug_on(objsvc_error)
    // Note: Do not include a semicolon after these two lines.
```

If the example functions are spread across multiple files, the above definitions must only be included in one file. The other files need to contain the following code lines:

```
    extern Debug objsvc_test;
    extern Debug objsvc_error;
```

The debug flags can be enabled using the following commands:

```
/opt/SUNWconn/em/bin/em_debug "on objsvc_test"
/opt/SUNWconn/em/bin/em_debug "on objsvc_error"
```

or alternatively using the following command:

```
/opt/SUNWconn/em/bin/em_debug "on objsvc_*"
```

6.5.2 `get_sys_dn` *Function*

Code Example 6-21 `get_sys_dn` Function

```
// Function to encode a distinguished name (TAG_CONT(2) for
// an instance of the system managed object class. The encoding
// assumes that the instance is contained directly under root
Oid       sys_id((char *)"2.9.3.2.7.4"); // ISO DMI systemId OID

Result
get_sys_dn(const char *sys_nm, Asn1Value &sys_fdn)
{
    Asn1Value sys_rdn;
    Asn1Value sys_ava;
    Asn1Value sys_name;
    Asn1Value sys_oid;

    if (!sys_nm)
        return NOT_OK;

    VTRY{
        TTRYRES(sys_fdn.start_construct(TAG_CONT(2)));
```

Code Example 6-21 `get_sys_dn` Function

```
// Function to encode a distinguished name (TAG_CONT(2) for
        TTRYRES(sys_rdn.start_construct(TAG_SET));
        TTRYRES(sys_ava.start_construct(TAG_SEQ));
        TTRYRES(sys_oid.encode_oid(TAG_OID, sys_id));
        TTRYRES(sys_name.encode_octets(TAG_GRAPHSTR, sys_nm));
        TTRYRES(sys_ava.add_component(sys_oid));
        TTRYRES(sys_ava.add_component(sys_name));
        TTRYRES(sys_rdn.add_component(sys_ava));
        TTRYRES(sys_fdn.add_component(sys_rdn));
    }
    VBEGHANDLERS
    VCATCHALL
        objsvc_test.print("get_sys_dn: error encoding DN\n");
        return NOT_OK;
    VENDHANDLERS

    return OK;
}
```

6.5.3 `get_graphstr_rdn` *Functions*

Code Example 6-22 `get_graphstr_rdn` Functions

```
// Function to encode an RDN consisting of an object identifier
(OID)
// and an ASN.1 GraphicString
Result
get_graphstr_rdn(const char *a_oidstr, const char *a_str,
Asn1Value &a_rdn)
{
    Asn1Value a_ava;
    Asn1Value a_oid;
    Asn1Value a_val;

    if (!a_oidstr || !a_str)
        return NOT_OK;

    TRY {
        TTRYRES(a_rdn.start_construct(TAG_SET));
        TTRYRES(a_ava.start_construct(TAG_SEQ));
        TTRYRES(a_oid.encode_oid(TAG_OID, Oid(a_oidstr)));
        TTRYRES(a_val.encode_octets(TAG_GRAPHSTR,
            DataUnit(a_str)));
        TTRYRES(a_ava.add_component(a_oid));
```

Code Example 6-22 `get_graphstr_rdn` Functions

```
// Function to encode an RDN consisting of an object identifier
(OID)
        TTRYRES(a_ava.add_component(a_val));
        TTRYRES(a_rdn.add_component(a_ava));
    }
    VBEGHANDLERS
    VCATCHALL {
        objsvc_test.print("\nError encoding RDN for %s\n",
a_val);
        return NOT_OK;
    }
    VENDHANDLERS

    return OK;
}

// Function to encode an RDN consisting of an object identifier
(OID)
// and an ASN.1 GraphicString
Result
get_graphstr_rdn(const Oid &a_oidval, const char *a_str,
Asn1Value
    &a_rdn)
{
    Asn1Value a_ava;
    Asn1Value a_oid;
    Asn1Value a_val;

    if (!a_str)
        return NOT_OK;

    TRY {
        TTRYRES(a_rdn.start_construct(TAG_SET));
        TTRYRES(a_ava.start_construct(TAG_SEQ));
        TTRYRES(a_oid.encode_oid(TAG_OID, a_oidval));
        TTRYRES(a_val.encode_octets(TAG_GRAPHSTR,
            DataUnit(a_str)));
        TTRYRES(a_ava.add_component(a_oid));
        TTRYRES(a_ava.add_component(a_val));
        TTRYRES(a_rdn.add_component(a_ava));
    }
    VBEGHANDLERS
    VCATCHALL {
```

Code Example 6-22 `get_graphstr_rdn` Functions

```
// Function to encode an RDN consisting of an object identifier
(OID)
        objsvc_test.print("\nError encoding RDN for %s\n",
a_val);
        return NOT_OK;
    }
    VENDHANDLERS

    return OK;
}
```

Topology API 7

7.1 Introduction

The Topology API is designed for use by Solstice Enterprise Manager application developers. This interface:

- Hides the topology implementation from application developers
- Allows applications to be more easily ported to future releases of EM as the PMI evolves and / or changes
- Allows for faster development of topology-based applications

The Topology API consists of the following classes:

Table 7-1 Topology API Classes

Class	**Description**
EMStatus Class	Reports status, including error
EMIntegerSet Class	Implements a general purpose integer set
EMIntegerSetIterator Class	Provides a convenient method to visit each member of the integer set
EMTopoPlatform Class	Represents the Topology API as a whole
EMObject Class	Specifies the interface supported by all the persistent object classes
EMTopoNodeDn Class	Identifies one topology node out of a set of topology node objects
EMTopoTypeDn Class	Identifies one topology type out of the set of topology types
EMTopoType Class	Represents a topology type
EMAgent Class	Contains the agent interface common between EMCmipAgent, EMRpcAgent, and EMSnmpAgent
EMCmipAgentDn Class	Identifies one rpc agent object out of the set of rpc agent objects
EMCmipAgent Class	Represents the MIS object which contains configuration information
EMRpcAgentDn Class	Identifies one rpc agent object out of the set of rpc agent objects
EMRpcAgent Class	Represents the MIS object which contains configuration information
EMSnmpAgentDn Class	Identifies one snmp agent object out of the set of snmp agent objects
EMSnmpAgent Class	Represents the MIS object which contains configuration information

7.2 Overview

While application developers must still work within the hierarchical model with containers and objects, they need not understand the multiple objects within the MIS that represent individual topology elements.

This API is not intended to provide access into non-topology related features provided by the PMI or Nerve Center Interface. Applications such as the Topology Import/Export Tool, Discover, and large parts of the Viewer should be achievable with this interface.

Note – Some understanding of GDMO/ASN.1 and network management principles is required to intelligently use the Topology API. For example, it is necessary to understand object identifiers (OID) and distinguished names (DN) and how these relate to the object registration tree and management information tree (MIT). In addition, a high-level understanding of the EM architecture and topology model is necessary.

- Standard RogueWave Tools.h++ classes and templates are used instead of the PMI's `DataUnit`, `Morf`, `Array`, and `Queue` classes. This is because the RogueWave Tools.h++ classes are simpler and easier to understand than the corresponding PMI classes. Furthermore, many developers just starting to work with Enterprise Manager are already familiar with the Tools.h++ library.
- Fundamental concepts such as accessing topology objects distributed across multiple MISs, and handling duplicate topology node names, have been factored into the design of the Topology API to simplify dealing with these issues.
- In general, the number of lines of code needed to perform some operation on topology objects are fewer (sometimes many times fewer) with the Topology API versus the PMI. In addition, the code is more readable and maintainable. This should allow for faster development of topology based applications.

7.3 General Description

Using the Topology API, developers can create applications for the EM platform without learning the details of the MIT naming tree. The following figure gives an idea of how the Topology API is positioned in EM.

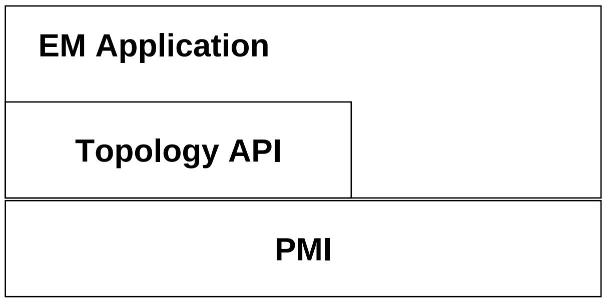


Figure 7-1 Position of the Topology API

7.4 Class Overview

Topology classes are related to the GDMO, the PMI, and persistent objects

7.4.1 Relationship to the GDMO

In terms of the GDMO, the Topology API provides a concrete C++ interface to the MIT objects described below:

Table 7-2 Topology API and GDMO Object Relationship

Topology API C++ Class	**Objects in MIT**
EMTopoNode	topoNode objects contained under topoNodeDBId=NULL topoView objects contained under topoViewDBId=NULL topoViewNode objects contained under topoViewDBId=NULL/topoNodeId=XX
EMTopoType	topoType objects contained under topoTypeDBId=NULL
EMCmipAgent	cmipAgent objects contained under agentTableType=”CMIP”
EMRpcAgent	rpcAgent objects contained under agentTableType=”RPC”
EMSnmpAgent	cmipsnmpProxyAgent objects contained under internetClassId={ 1 3 6 1 4 1 42 2 2 2 9 2 4 1 0}

Through the C++ interface, the Topology API provides services that are equivalent to the following GDMO services:

- Creation, deletion, set attributes of, and get attributes of the object classes topoNode, topoView, topoViewNode, topoType, cmipsnmpProxyAgent, cmipAgent, and rpcAgent.
- The actions topoNodeGetByName, topoNodeGetByType, topoNodeGetByMO, and topoGetViewGraph supported by the topoNodeDB object class.
- objectCreation, objectDeletion, and attributeValueChange notifications for the topology object classes topoNode, topoView, topoViewNode.

7.4.2 Relationship to the PMI

The Topology API is built on top of the PMI. If your client application only needs to manipulate topology nodes, topology types, cmip agents, rpc agents, and snmp agents, then the only places where the PMI must still be used are the following:

- A connection to an MIS, established using the PMI `Platform` class. After the platform instance has been successfully initialized, the Topology API is initialized by calling the EMTopoPlatform::initialize method with the Platform instance as a parameter.
- If the application supports access control application features, then Platform::get_authorized_features() must be used to find out which features a particular user is authorized to use.
- The `Morf` class, used for setting or getting the EMTopoNode::user_data attribute. There is really no way around this, since the user_data attribute can contain data defined by any ASN.1 syntax.

Of course, if the application needs to access additional objects in the MIT beyond those outlined in Table 7-2 on page 7-4, then the PMI Image class must be used. As an example, the Solstice EM Viewer makes use of the Topology API to access topology nodes and topology types, the NCI API to manipulate requests, and the PMI for connection, access control features, and a few misc. operations.

The following restrictions on usage of the PMI are necessary in order for the Topology API to function correctly:

If Platform::replace_discriminator() or Platform::replace_discriminator_classes to eliminate the sending of unwanted events from the MIS, the discriminator must allow all events for the following object classes: topoNode, topoView, topoViewNode, topoType, cmipAgent, rpcAgent, and cmipsnmpProxyAgent. For example, an application which doesn't subscribe for any events itself, i.e. doesn't use Album::when(), Image::when(), or Platform::when(), call Platfomr::replace_discriminator() with the argument "or : {} " which instructs the MIS to not send any events to the application, thus eliminating unnecessary event traffic and processing time. However, if that same application was using the Topology API, then the following would have to be done::

```
#include <pmi/hi.hh>

Array(DU) object_clases;
object_classes.alloc(7);
object_classes[0] = "topoNode";
object_classes[1] = "topoView";
object_classes[2] = "topoViewNode";
object_classes[3] = "topoType";
object_classes[4] = "cmipAgent";
object_classes[5] = "rpcAgent";
object_classes[6] = "cmipsnmpProxyAgent";

Platform::replace_discriminator_classes(object_classes)
```

- Platform::set_attr_coder() should not be called to change the encoder / decoder for any of the GDMO attributes of the GDMO object classes topoNode, topoView, topoViewNode, topoType, cmipAgent, rpcAgent, or cmipsnmpProxyAgent.

7.4.3 `EMTopoPlatform` *Class*

The `EMTopoPlatform` class represents the Topology API as a whole. Only one instance of the `EMTopoPlatform` class is allowed. This instance is initialized by calling `EMTopoPlatform::initialize()`, and is accessed through the `EMTopoPlatform::instance()` method[1]. The `EMTopoPlatform` class provides various methods, including:

- Get all MIS systems reachable from the connected MIS

1. For those familiar with C++ / OO design, the `EMTopoPlatform` class uses the Singleton pattern.

- Find topology nodes by name, type, or managed object
- Find CMIP, RPC, and SNMP agents by managed object.
- Get the topology pathname(s) by topology node DN.

```
#include <pmi/hi.hh>
#include <topo_api/topo_api.hh>

Platform platform;

if (!platform.connect("mishost","em_client")) {
cerr << "Failed to connect to " << "mishost" << endl;
exit(-1);
}
EMTopoPlatform::initialize(platform);
```

The above code shows how the Topology API is initialized.

7.4.4 *Persistent Object Classes*

The Topology API provides an interface to five persistent objects in the MIS: topology nodes, topology types, SNMP agents, CMIP agents, and RPC agents. Unlike the PMI, where the `Image` class provides a generic interface to any persistent object in the MIS, the Topology API provides a concrete, type-safe C++ class for each of these five classes.

7.4.4.1 `EMObject` *Class*

The `EMObject` class is an abstract base class from which the concrete POC classes are derived. This class declares the common methods that all POC classes support. The common methods include:

- Creation and deletion in the MIS's persistent store.
- Loading/storing attributes from/to the MIS

Other methods common among POC classes are not declared in the `EMObject` class because the signature of the method isn't exactly the same. For example, all the POC classes support the `compare_all_attributes()` and `compare_some_attributes()` methods. For the `EMTopoNode` POC class, the method signatures are:

```
RWBoolean EMTopoNode::compare_all_attributes(const EMTopoNode&
peer);

RWBoolean EMTopoNode::compare_some_attributes(const EMTopoNode&
peer);
```

Because the type of the <*peer*> parameter differs for each POC, the methods cannot be included in the `EMObject` base class. Each POC class also provides additional methods specific to the POC; in particular, access methods to the attributes of the POC are provided.

7.4.4.2 `EMTopoType` *Class*

An instance of the EMTopoType class represents a topology type. Every topology node is classified as a particular topology type. The topology types form a hiearchy with the four base types “Container”, “Device”, “Monitor”, and “Link” with other subtypes derived from them. Beyond the standard POC methods which allow you to create, delete, compare, etc. topology types, the EMTopoType class provides the following additional services:

- static methods is_container(), is_device(), is_monitor(), is_link(), and is_view() (equivalent to is_monitor() or is_container()) can be used to categorize topology types.

7.4.4.3 `EMTopoNode` *Class*

The `EMTopoNode` class represents a topology node, which is the unit of management in Solstice EM. Using the standard POC methods, you can create, delete, and compare topology nodes. Using the EMTopoNode's access methods you can get and set the name, topology pathname, logical and geographical location, topology type, and associated managed objects and their corresponding CMIP, RPC, and/or SNMP agent objects among others attributes. The EMTopoNode class also provides a callback mechanism to notify clients when a topology node has been created, deleted, or had one or more attributes changed.

7.4.4.4 `EMSnmpAgent` *Class*

An instance of the EMSnmpAgent class represents the MIS object which contains configuration information for an SNMP agent. The configuration information includes the read and write community strings, supported MIBs, and transport address. NOTE: This class does not provide an interface to the agent's managed objects, but only to Solstice EM's configuration information for the agent.

7.4.4.5 `EMCmipAgent` *Class*

An instance of the EMCmipAgent class represents the MIS object which contains configuration information for an CMIP agent. The configuration information includes the CMIP MPA hostname and port number, list of managed objects DNs, network SAP, transport selector, presentation selector, session selector, and application entity title (AET). NOTE: This class does not provide an interface to the agent's managed objects, but only to Solstice EM's configuration information for the agent.

7.4.4.6 `EMRpcAgent` *Class*

An instance of the EMRpcAgent class represents the MIS object which contains configuration information for an RPC agent. The configuration information includes the read and write community strings, and supported schemas. NOTE: This class does not provide an interface to the agent's managed objects, but only to Solstice EM's configuration information for the agent.

7.4.5 Utility Classes

7.4.5.1 `EMIntegerSet` Class

The EMIntegerSet class implements a general-purpose integer set over the numbers 0 to n. It is used in the Topology API to communicate which attributes of a PIC that an API method should operate on. The example below shows how to load only the name and topology type of a topology node.

7.4.5.2 `EMStatus` Class

Instances of class EMStatus are returned by almost every API method to report status, including errors. A conversion operator to RWBoolean is provided so that EMStatus can be evaluated in boolean expressions. A value of FALSE means there was an error, otherwise success. The following sample code shows the basic usage.

7.5 Topology API Concepts

7.5.1 Element Naming

Applications must be able to access individual topology elements without traversing the entire topology hierarchy. The mapping of topology element names to that of a file system model for unique naming is supported. In the event that a file system style reference is ambiguous within the underlying MIT, the method invoked fails and reports the appropriate error. As an example of this naming, an element named “Parrothead,” located under the “Internet” view, located under the “Root” view, would be referenced as `/Root/Internet/Parrothead`. There can only be one root and as such, the root is represented as `/` within this model.

7.5.2 Duplicate Topology Node Names

The administrative names of the EMTopoType, EMCmipAgent, EMSnmpAgent, and EMRpcAgent persistent objects are guaranteed to be unique. In contrast, the administrative name of the EMTopoNode is not guaranteed to be unique.

To address this, the `EMTopoPlatform` class provides several methods to return a list of EMTopoNodeDn instances that:

- Have the same administrative name
- Have the same type
- Share the same proxy agent object
- Share the same managed object dns

7.5.3 MIS-MIS Awareness

Each persistent object class supports access to any object instance visible from the connected MIS. For example, if MIS A and MIS B have a 2-way MIS-MIS connection setup, you can connect to MIS A, then modify EMTopoNodes, EMTopoTypes, and so forth, on MIS B.

The `EMTopoPlatform` find methods, such as `find_topo_nodes_by_name()`, and `find_topo_nodes_by_type()`, perform the search on the entire set of objects visible from the connected MIS. Using the above example of MIS A and MIS B, if you connect to MIS A, and `find_topo_nodes_by_type()`, you see a list of all `EMTopoNodeIds` on MIS A and/or MIS B of the indicated type.

7.5.4 Performance Considerations

Because the Topo API is built on top of the PMI, most operations take slighter longer when using the Topo API versus writing the code directly with PMI.

In terms of memory usage, however, the persistent object classes require much less memory cache information about an object than if an `Image` class had been used instead. This is because the Topo API classes can optimize the data storage; they know exactly what attributes each managed object contains.

7.6 Examples

This section presents several examples showing how to use the Topology API for common tasks.

7.6.1 Makefile

The following Makefile was used to compile all of the programs in this section. The version of the SUN C++ SparcCompiler used is “SC4.0 18 Oct 1995 C++ 4.1”. (This is the output from “CC -V”). This Makefile and the following sample programs can be found in $EM_HOME/src/topo_api directory.:

```
CCFLAGS = +w -g -noex -I${EM_HOME}/include
-I${EM_HOME}/include/pmi
LDFLAGS =  -L${EM_HOME}/lib -ltopo_api -lpmi -lrwtool -lsched
-lnsl -lsocket -lgen -R/opt/SUNWconn/em/lib


EXES =print_topo topo_events traverse
OBJS =$(EXES:%=%.o)

all: $(EXES)

print_topo: print_topo.o
    $(LINK.C) -o $@ $@.o

topo_events: topo_events.o
    $(LINK.C) -o $@ $@.o

traverse: traverse.o
    $(LINK.C) -o $@ $@.o

clean:
    rm -rf $(EXES) $(OBJS) Templates.DB;
```

7.6.2 Finding Topology Nodes

This program accepts as input the name of a topology node(s). The program then uses EMTopoPlatform::find_nodes_by_name() to find all topology nodes with the given name. Then some information for each node is printed out. This program highlights the fact that more than one topology node can have the same name, and so a Solstice EM client should never assume that the topology node names are unique. That is why the Topology API uses instances of EMTopoNodeDn to uniquely identify a single topology node.

.

```
#include <stdio.h>
#include <topo_api/topo_api.hh>

int
main(int argc, char**argv)
{
if (argc < 2) {
       cerr << “Usage: “ << argv[0] << “ node-name” << endl;
       exit(-1);
}
RWCString node_name = argv[1];

Platform platform(duEM);

if (!platform.connect(““,”em_sample”)) {
       cerr << “Couldn’t connect!” << endl;
       exit (-2);
}

EMTopoPlatform::initialize(platform);

RWTValSlist<EMTopoNodeDn> nodes;
    EMTopoPlatform::instance()-
>find_nodes_by_name(node_name,nodes);

if (nodes.isEmpty()) {
       cerr << “No Topology Node Named “ << node_name << endl;
       exit(-3);
}

for (RWTValSlistIterator<EMTopoNodeDn> i(nodes); i(); ) {
       EMTopoNode node(i.key());
       EMStatus status;

       if (!(status = node.load_all_attributes())) {
           cerr << “Error: “ << status << endl;
           exit(-4);
       }
```

```
        //
        // The stream output operator << is defined for
        // EMTopoNode, EMTopoType, EMCmipAgent, EMSnmpAgent,
        // EMRpcAgent, providing an easy way to print out
        // the values of an objects while debugging.
        cout << “---------debug output--------” << endl;
        cout << node << endl;
        cout << “---------debug output--------” << endl;

        //
        // Normally, you want to do more with the values than
        // print them out.
        //
        EMTopoNode::Severity severity;
        node.get_severity(severity);

        RWCString name;

        node.get_name(name); // name should be the same as
node_name

        RWCString type_name;
        node.get_type_name(type_name);

        EMTopoNode::GeoLocation geographical_location;
        RWBoolean is_geographical_location_null;
        node.get_geographical_location(geographical_location,
                       is_geographical_location_null);

        RWTValSlist<EMCmipAgentDn> cmip_agents;
        RWTValSlist<EMSnmpAgentDn> snmp_agents;
        RWTValSlist<EMRpcAgentDn> rpc_agents;
        node.get_cmip_agents(cmip_agents);
        node.get_snmp_agents(snmp_agents);

        node.get_rpc_agents(rpc_agents);

        cout << “Node named “ << name << “ is of type “ <<
type_name << endl
            << “The most severe outstanding alarm is “ << severity
<< “.” << endl
```

```
            << “The node is located at “;
        if (is_geographical_location_null)
            cout << “<unknown>”;
        else
            cout << geographical_location;
        cout << “ in the world” << endl;

        cout << “CMIP agents: “;
        if (cmip_agents.isEmpty()) {
            cout << “none” << endl;
        } else {
            cout << endl;
            for (RWTValSlistIterator<EMCmipAgentDn>
j(cmip_agents); j(); ) {
                cout << “\t” << j.key() << endl;
            }
        }

        cout << “RPC agents: “;
        if (rpc_agents.isEmpty()) {
            cout << “none” << endl;
        } else {
            cout << endl;
            for (RWTValSlistIterator<EMRpcAgentDn> j(rpc_agents);
j(); ) {
                cout << “\t” << j.key() << endl;
            }
        }

        cout << “SNMP agents: “;
        if (snmp_agents.isEmpty()) {
            cout << “none” << endl;
        } else {
            cout << endl;
            for (RWTValSlistIterator<EMSnmpAgentDn>
j(snmp_agents); j(); ) {
                cout << “\t” << j.key() << endl;
            }
        }
    }
return 0;
}
```

7.6.3 Registering Events for `EMTopoNode`

The EMTopoNode class provides an event subscription service to notify clients when a topology node is created, deleted, or modified. This service is not offered by the other persistent object classes.

The following program registers for all three types of events and then proceeds to create, modify, and then destroy a single topology node in order to cause some events to be sent to the registered callback. A description of the each event is printed to stdout.

```
#include <stdio.h>
#include <topo_api/topo_api.hh>

void topo_event_cb(
    const EMTopoNodeCallbackData& cbd
);


int
main(int /*argc*/, char** /*argv*/)
{
    Platform platform(duEM);

    if (!platform.connect("","em_sample")) {
    cerr << "Couldn't connect!" << endl;
    exit (-2);
    }

    EMTopoPlatform::initialize(platform);

    //
    // Register for create, delete, and attribute change events
    // on EMTopoNode objects.
    //

EMTopoNode::register_callback(em_any_event,topo_event_cb,NULL);

    //
    // Now we will create, modify, and then delete an EMTopoNode
    // to trigger some events.
    //

    //
    // Find the root node(s) (there will one for each MIS)
    // so we have a parent view to create a topology node in.
    //
    RWTValSlist<EMTopoNodeDn> roots;
    EMTopoPlatform::instance()->find_root_nodes(roots);
}
```

```
//
    // Okay, now we have to set the three mandatory attributes
   // for creating a topology node: name, type_name, and parents.
    //
    EMTopoNode node;

    node.set_name("first-name");

    node.set_type_name(EMTopoTypeDn::host);
    //
    // Arbitrarily use the first root node in the list as the
    // parent of the new topology node.
    //
    node.add_parent(roots.first());

    //
    // Create the node
    //

    EMStatus status;
    if (!(status = node.create_with_all_attributes())) {
    cerr << "Error: " << status << endl;
    exit(-1);
    }

    //
    // After the create has completed, the EMTopoNode::dn
    // attribute, the unique identifier, is set.
    //
    EMTopoNodeDn node_dn;
    node.get_dn(node_dn);
    cout << "Created Topology Node Id=" << node_dn <<
    " with parent Id=" << roots.first() << endl;

    //
    // Modify some attributes
    //
   // We don't want to store the parents and type_name attributes
    // again, so we need to reset the EMTopoNode object.
    node.clear_all_attributes();
```

```
 //
    // EMTopoNode::dn is the only mandatory attribute when doing
    // a store or load.
    //
    node.set_dn(node_dn);
    node.set_name("second-name");

node.set_logical_location(roots.first(),Location(/*x=*/12,/*y=*
/34,/*z=*/56));
    node.set_geographical_location(GeoLocation(/*longitude=*/-
112.0,/*latitude=*/45.0));

    //
    // store_all_attributes() only stores attributes that
    // have been set.
    //
    if (!(status = node.store_all_attributes())) {
    cerr << "Error: " << status << endl;
    exit(-2);
    }

    //
    // destroy the node
    //
    if (!(status = node.destroy())) {
    cerr << "Error: " << status << endl;
    exit(-3);
    }
}

void topo_event_cb(
    const EMTopoNodeCallbackData& cbd
)
{
    switch(cbd.event_type) {
      case em_create_event:
        cout << "topo_event_cb: node " << cbd.node_dn << "
created" << endl;
        break;
      case em_delete_event:
        cout << “topo_event_cb: node “ << cbd.node_dn << “
deleted” << endl;
        break;
      case em_change_event:
        cout << “topo_event_cb: node “ << cbd.node_dn << “
modifed” << endl;

        //
        // cbd.changes is an instance of EMTopoNode which contains
all of
        // the changes
        //
        EMIntegerSet attributes(EMTopoNode::num_attributes);
        cbd.changes.get_active_attributes(attributes);
        cout << “\tAttributes changed: “;
        for (EMIntegerSetIterator i(attributes); i.next(); ) {
            cout << EMTopoNode::get_attribute_name(i.member());
        }
        cout << endl;

        for (i.reset(); i.next(); ) {
            switch (i.member()) {
                case EMTopoNode::name:
                    {
                        RWCString name;
                        cbd.changes.get_name(name);
                        cout << “\tname changed to “ << name <<
endl;
                    }
                    break;
                case EMTopoNode::logical_locations:
                    {
                        RWTValSlist<LocationInParent> locations;

cbd.changes.get_logical_locations(locations);
                        cout << “\tlogical_location in parent view
“ << locations.first().parent <<
                        “ is “ << locations.first().location <<
endl;
                case EMTopoNode::children:
                   {
                       RWTValSlist<EMTopoNodeDn> children;
                       cbd.changes.get_children(children);

                       cout << “\tchildren changed to {“;
                       for (RWTValSlistIterator<EMTopoNodeDn>
j(children); j(); ) {
                           cout << j.key() << “ “;
                       }
                       cout << “}” << endl;
                   }
                   break;
           }
       }
   }
}
```

7.6.4 Printing the Topology Hierarchy

The topology hiearchy forms a directed acyclic graph. It is a graph, rather than a tree, because each topology node except the root node can have more than one parent. It is acyclic because the parent-child relationship should not have any loops. The following program traverses the topology depth-first starting at the root node(s). As each node is visited, the EMTopoNode:topology_pathnames attribute is printed to stdout. Note that the <cache_view_graph> option of EMTopoPlatform::initialize() is set to TRUE this time since the program makes accesses the EMTopoNode::topology_pathnames attribute of every node.

```
#include <stdio.h>
#include <iostream.h>
#include <topo_api/topo_api.hh>

void traverse(
    const EMTopoNodeDn& dn
);

long num_traversed = 0;

int
main(int /*argc*/, char** /*argv*/)
{
    Platform platform(duEM);

    if (!platform.connect("localhost","em_sample")) {
        cerr << "Couldn't connect!" << endl;
        exit (-1);
    }

    EMTopoPlatform::initialize(platform,TRUE);

    EMStatus status;

    //
    // Find the root node(s) (there will one for each MIS)
    // so we have a parent view to create a topology node in.
    //
    RWTValSlist<EMTopoNodeDn> roots;
    if (!(status = EMTopoPlatform::instance()-
>find_root_nodes(roots))) {
        cerr << status << endl;
        exit(-2);
    }
```

```
 //
    // Traverse the topology of each MIS
    //
    for (RWTValSlistIterator<EMTopoNodeDn> i(roots); i();) {
        traverse(i.key());
    }

    //
   // Note that the number of nodes traversed is most likely not
equal
   // to the number of nodes since a node with n parents would be
    // traversed n times with the simple algorithm used.
    //
    cout << "Num Nodes Traversed = " << num_traversed << endl;
}

void
traverse(
    const EMTopoNodeDn& dn
)
{
    EMStatus status;

    EMTopoNode node(dn);

    num_traversed++;

    //
   // Load the topology pathnames of the topology node, and also
    // the children if the topology node is a view (container or
    // monitor type)
    //
    EMIntegerSet attributes(EMTopoNode::num_attributes);
    attributes.add(EMTopoNode::topology_pathnames);

    // We need the children attribute so that we can continue
    // or depth-first traversal.
    if (EMTopoPlatform::instance()->is_view(dn)) {
        attributes.add(EMTopoNode::children);
    }
 if (!(status = node.load_some_attributes(attributes))) {
        cerr << “Error: “ << status << endl;
        exit(-3);
    }

    //
   // Print out all the possible topology pathnames for the node.
    // A node may have more than one valid pathname from the root
    // node because this is a directed acyclic graph not a tree.
    //

    RWTValSlist<RWCString> topology_pathnames;
    if (!(status =
node.get_topology_pathnames(topology_pathnames))) {
        cerr << “Error: “ << status << endl;
        exit(-3);
    }

    cout << “{ “;
    for (int i = 0; i < topology_pathnames.entries(); i++) {
        cout << topology_pathnames[i];
        if (i !=  topology_pathnames.entries() - 1)
            cout << “, “;
    }
    cout << “}”<< endl;

    //
    // Recur on the node’s children (if it has any)
    //
    if (EMTopoPlatform::instance()->is_view(dn)) {
        RWTValSlist<EMTopoNodeDn> children;
        if (!(status = node.get_children(children))) {
            cerr << “Error: “ << status << endl;
        }
        for (RWTValSlistIterator<EMTopoNodeDn> j(children); j();)
{
            traverse(j.key());
        }
    }
}
```

7.7 Class Reference

This section includes the following classes:

- *EMStatus Class*
- *EMIntegerSet Class*
- *EMIntegerSetIterator Class*
- *EMTopoPlatform Class*
- *EMObject Class*
- *EMTopoNodeDn Class*
- *EMTopoTypeDn Class*
- *EMTopoType Class*
- *EMAgent Class*
- *EMCmipAgentDn Class*
- *EMCmipAgent Class*
- *EMRpcAgentDn Class*
- *EMRpcAgent Class*
- *EMSnmpAgentDn Class*
- *EMSnmpAgent Class*

7.7.1 `EMStatus` *Class*

Inheritance:

```
#include <topo_api/topo_api.hh>
```

Description

Instances of class EMStatus are returned by almost every API method to report status, including errors. A conversion operator to RWBoolean is provided so that EMStatus can be evaluated in boolean expressions. A value of FALSE means there was an error, otherwise success. The following sample code shows the basic usage.

Static Variables

```
static EMStatus EMStatus::success;
```

This static public member can be used to compare to instances of EMStatus. If they are equal, then the operation succeeded. For example:

```
if (EMStatus::success == node.load_all_attributes()) {
    cout << “succeeded” << endl;
}
```

Enumerations

```
enum EMStatus::Code {
        successful,
        pmi_error,
        object_doesnt_exist,
        attribute_is_not_creatable,
        attribute_is_not_storeable,
        attribute_not_set,
        key_not_found,
        missing_mandatory_attribute,
        cannot_set_attribute,
        decode_error,
        encode_error,
        attribute_not_registered,
        does_not_exist,
        already_exists,
        invalid_arg,
        not_implemented,
        not_supported,
        view_graph_not_cached,
        duplicate_cmip_managed_fdns,
        unknown_error, /* this means an internal error*/
        num_status
    };
```

These are all the possible statuses that can be returned by the API.

Constructors, Destructor, Assignment Operator

```
EMStatus();
EMStatus(
    Code error_code,
    const RWCString& text
);

EMStatus(
    const EMStatus& status
);
~EMStatus();
EMStatus& operator =(
    const EMStatus& status
);
```

Normally, only the default constructor is used by clients.

Operators

```
RWBoolean operator ==(
    const EMStatus& status
) const;

RWBoolean operator !=(
    const EMStatus& status
) const;

operator RWBoolean () const;
```

The RWBoolean conversion operator equates EMStatus::successful with TRUE, otherwise FALSE. Here is an example of how the this can be used to test the return status of a method:

```
EMStatus status;
if (!(status = node.load_all_attributes())) {
    cerr << "Error: " << status << endl;
}
```

```
EMStatus::Code code() const;
```

Returns the status code.

Global Operators

```
ostream& operator<<(
    ostream& s,
    const EMStatus& status
);
```

The stream output operator << is defined to provide an easy was to print out the value of EMStatus.

7.7.2 `EMIntegerSet` *Class*

Inheritance: none

```
#include <topo_api/topo_api.hh>
```

Description

The EMIntegerSet class implements a general-purpose integer set over the numbers 0 to n. It is used in the Topology API to communicate which attributes of a PIC that an API method should operate on. The example below shows how to load only the name and topology type of a topology node.

Example

Instances of EMIntegerSet are used when you only want to load, store, or compare a subset of the attributes of one of the persistent object classes.

```
EMIntegerSet attrs(EMTopoNode::num_attributes);
attrs.add(EMTopoNode::name);
attrs.add(EMTopoNode::type_name);
attrs.add(EMTopoNode::parents);

node.load_some_attributes(attrs);
```

Constructors, Destructor, Assignment Operator

```
EMIntegerSet ();
EMIntegerSet(
     long n
);
EMIntegerSet(
     long n,
     RWBoolean initVal
);
EMIntegerSet& operator = (
     RWBoolean b
);
```

EMIntegerSet(n) constructs a set of integers drawn from the universe numbered from 0 to n-1. By default, no numbers in the universe belong to the set. EMIntegerSet(n,TRUE) is the same except that every integer is a member of the set.

Operators

```
RWBoolean operator == (
     const EMIntegerSet& set
) const;

RWBoolean operator != (
     const EMIntegerSet& set
) const;
```

Two EMIntegerSet instances are considered equal if they are both of the same dimension <n> and have the exact same members.

```
RWBoolean operator [] (
     long number
) const;
```

Returns TRUE if the integer <n> is a member of the set.

```
    EMIntegerSet& operator&=(
        const EMIntegerSet& set
    );

    EMIntegerSet& operator^=(
        const EMIntegerSet& set
    );

    EMIntegerSet& operator|=(
        const EMIntegerSet& set
    );
```

Peforms a member-wise and, exclusive-or, or or boolean operation on the set with another set of integers which must be of the same dimension.

Methods

```
    void add(
        long number
    );

    void remove(
        long number
    );
```

Adds and removed integers from the set.

```
    RWBoolean is_member(
        long number
    ) const;
```

Returns TRUE if <number> is a member of the set.

```
    long num_members() const;

    long max_members() const;
```

Returns the number of integers in the set and the number of integers in the universe of potential members.

```
    void resize(long n);
```

Resizes the integer set to the universe of integers 0 to n-1.

Global Operators

```
    EMIntegerSet operator!(
        const EMIntegerSet& set
    );

    EMIntegerSet operator&(
        const EMIntegerSet& set1,
        const EMIntegerSet& set2
    );

    EMIntegerSet operator^(
        const EMIntegerSet& set1,
        const EMIntegerSet& set2
    );

     EMIntegerSet operator|(
        const EMIntegerSet& set1,
        const EMIntegerSet& set2
    );
```

The operator! returns the member-wise negation of input set. The other functions return the member-wise and, exclusive-or, and or of two sets. Note that for the binary operations, the two sets must be of the same dimensions.

7.7.3 `EMIntegerSetIterator` *Class*

Inheritance: none

```
#include <topo_api/topo_api.hh>
```

Description

The `EMIntegerSetIterator` class provides a convenient method to visit each member of the integer set.

Example

```
EMTopoNode node;
EMItegerSet(EMTopoNode::num_attributes);
node.get_active_attributes(attrs);
for (EMIntegerSetIterator i(atrrs); i(); ) {
    switch (i.key()) {
            case EMTopoNode::name:
                break;
            case EMTopoNode::type_name:
                break;
            case EMTopoNode::managed_objects:
                break;
            default:
                cout << “Some other attribute” << endl;
                break;
    }
}
```

Constructors, Destructor, Assignment Operator

```
EMIntegerSetIterator(const EMIntegerSet& set);
```

The Iterater will visit the members of <set> in numerical order.

Methods

```
RWBoolean next();
long member() const;
void reset();
```

`next()` advances the iterater one position and returns TRUE if the new position is valid, FALSE otherwise. `member`() returns the integer member which is currently being visited. `reset()` resets the iterater to the first integer member in the set

7.7.4 `EMTopoPlatform` *Class*

Inheritance: none

```
    #include <topo_api/topo_api.hh>
```

Description

The `EMTopoPlatform` class represents the Topology API as a whole. Only one instance of the `EMTopoPlatform` class is allowed. This instance is initialized by calling `EMTopoPlatform::initialize()`, and is accessed through the `EMTopoPlatform::instance()` method. The `EMTopoPlatform` class provides various methods, including:

- Get all MIS systems reachable from the connected MIS
- Find topology nodes by name, type, or managed object
- Find CMIP, RPC, and SNMP agents by managed object.
- Get the topology pathname(s) by topology node DN.

Example

```
#include <pmi/hi.hh>
#include <topo_api/topo_api.hh>

Platform platform;
if (!platform.connect(“mishost”,”em_client”)) {
cerr << “Failed to connect to “ << “mishost” << endl;
exit(-1);
}
EMTopoPlatform::initialize(platform);

EMStatus status;
RWTValSlist<EMTopoNodeDn> root_nodes;
if (!(status = EMTopoPlatform::instance()->
           find_root_nodes(root_nodes))) {
    cerr << “Error: “ << status << endl;
    exit(-1);
}
```

The above code shows how the Topology API is initialized.

Static Methods

```
static RWBoolean initialize(
        Platform& platform,
        RWBoolean cache_view_graph = FALSE
    );

static EMTopoPlatform* instance();
```

initialize() must be called before using any of the Topology API classes. This method should only be called after the <platform> has been successfully initialized. It returns TRUE on success, FALSE otherwise.

The optional parameter <cache_view_graph> specifies whether should optimize methods which operate over the topology view hiearchy. If <cache_view_graph> is TRUE, then the topology view hierarchy will be cached into memory using from the MIS using a special GDMO action “topoGetViewGraph” on the “topoNodeDBId=NULL” object. This optimization greatly increases the speed of loading

EMTopoNode::view_children and EMTopoNode::topology_pathnames and executing EMTopoPlatform::is_view(). However, the view cache requires can require a significant amount of time and memory for large topology view hierarchy (> 5000 nodes). Table 7-3 summarizes when the <cache_view_graph> option should be turned on and off.

Table 7-3 cache_view_graph Option

<cache_view_graph>	**application type**
TRUE	Frequently calls EMTopoPlatform::is_view(), EMTopoPlatform::view_topology_pathams(), EMTopoPlatform::topology_pathnames(), and / or loads EMTopoNode::view_children, EMTopoNode::topology_pathnames attributes.
FALSE	Doesn't use the above features or uses them infrequently. Client application want to be as lightweight as possible and startup fast.

After initialize() succeeds, the static method instance() will return a pointer to EMTopoPlatform instance from which the non-static EMTopoPlatform methods can be invoked. See ***Example*** section above.

Access Methods

```
    const RWCString& local_system_name() const;

    RWTValSlist<RWCString> system_names() const;

    Platform& platform();
```

local_system_name() returns the name of the MIS that the client application connected to while system_names() returns a list of all MIS names which are visible through the connection to the local MIS, including the local MIS name. Note: For a remote MIS to be visible to client application, MMC (MIS-MIS Communication) must be setup between the local MIS and each remote MIS. This can be accomplished using the em_mismgr application.

General Methods

```
EMStatus find_root_nodes(
     RWTValSlist<EMTopoNodeDn>& root_nodes,
    const RWTValSlist<RWCString>& system_names =
        RWTValSlist<RWCString>()
) const;
```

Returns a list of all nodes named “Root” visible through the connection to the local MIS. The optional parameter <system_names> specifies the list of MISes to restrict the query. The MIS names in <system_names> should all appear in system_names(), otherwise EMStatus::invalid_arg will result. If <system_names> is empty (the default), then the list returned by system_names() is used.

```
EMStatus find_nodes_by_name(
     const RWCString& name,
     RWTValSlist<EMTopoNodeDn>& nodes,
    const RWTValSlist<RWCString>& system_names =
        RWTValSlist<RWCString>()
) const;
```

Same as find_root_nodes() except that all nodes named <name> are returned instead of all nodes named “Root”.

```
EMStatus find_nodes_by_type(
     const RWCString& type_name,
     RWTValSlist<EMTopoNodeDn>& nodes,
    const RWTValSlist<RWCString>& system_names =
        RWTValSlist<RWCString>()
) const;
```

Same as find_root_nodes() except that all nodes named of type <type_name> are returned instead of all nodes named “Root”.

```
EMStatus find_nodes_by_managed_object(
        const RWCString& managed_object,
        RWTValSlist<EMTopoNodeDn>& nodes,
       const RWTValSlist<RWCString>& system_names =
           RWTValSlist<RWCString>()
) const;
```

Same as find_root_nodes() except that all nodes which have <managed_object> listed in their EMTopoNode::managed_objects attribute are returned instead of all nodes named “Root”.

```
RWBoolean is_view(
        const EMTopoNodeDn& node_dn
) const;
```

Returns TRUE if the topology type of node <node_dn> is a ‘view’ type, i.e. a subtype of EMTopoTypeDn::container or EMTopoTypeDn::monitor.

```
EMStatus view_topology_pathnames(
        const EMTopoNodeDn& view_dn,
       RWTValSlist<RWCString>& pathnames
) const;
```

Returns in <pathnames> a list of all topology pathnames for the node <view>. The node <view> should be a ‘view’, i.e. a subtype of EMTopoType::container or EMTopoTypeDn::monitor. If <cache_view_graph> optimization is turned on, then this method is relatively inexpensive since no information needs to be retrieved from the MIS. At a minimum, <pathnames> will contain one pathname for each parent. However, since each parent can also have more than one parent, and so on, the actual number of pathnames may be higher.

```
    EMStatus topology_pathnames(
        const EMTopoNodeDn& parent_dn,
        const RWCString& name,
        RWTValSlist<RWCString>& pathnames
    ) const;
```

Similar to view_topology_pathnames() except that this version will work for any type of node. This method will also not send any data requests to the MIS if the <cache_view_graph> optimization is turned on. The reason that the parameters <parent_dn> and <name> are required rather than just the EMTopoNodeDn of the node is because these two pieces of information are not cached by the Topology API -- the topology view cache has this information but only for view nodes. By having the client application pass this information in, the Topology API can take advantage of the cases where the client already has this information in memory.

```
    EMStatus find_root_types(
        RWTValSlist<EMTopoTypeDn>& types
    ) const;
```

Returns in <types> a list of all root types, i.e. types who have no EMTopoType::base_type. In the default installation of Solstice EM, the root types are EMTopoTypeDn::container, EMTopoTypeDn::device, EMTopoTypeDn::monitor, and EMTopoTypeDn::link. NOTE: Unlike the find_root_nodes() method, this method will only return the root types on the local MIS.

```
    EMStatus find_all_types(
        RWTValSlist<EMTopoTypeDn>& types
    ) const;
```

Similar to find_root_types() except that <types> will contain all topology types on the local MIS.

```
    EMStatus managed_object_to_cmip_agent(
        const RWCString& managed_object,
        EMCmipAgentDn& cmip_agent
    ) const;

    EMStatus managed_object_to_snmp_agent(
        const RWCString& managed_object,
        EMSnmpAgentDn& snmp_agent
    ) const;

    EMStatus managed_object_to_rpc_agent(
        const RWCString& managed_object,
        EMRpcAgentDn& rpc_agent
    ) const;
```

These methods provide a method to find the agent which is responsible for a particular <managed_object>. If a match is not found, then EMStatus::key_not_found error will result. These methods are used internally by the EMTopoNode class to calculate the EMTopoNode::snmp_agents, EMTopoNode::rpc_agents, and EMTopoNode::cmip_agents attributes from the EMTopoNode::managed_objects attribute.

7.7.5 `EMObject` *Class*

Inheritance: none

```
#include <topo_api/topo_api.hh>
```

Description

The `EMObject` class is an abstract base class which specifies the interface supported by all the persistent object classes (POC): EMTopoNode, EMTopoType, EMCmipAgent, EMRpcAgent, and EMSnmpAgent.

Each POC instance is an interface to a particular set of objects in the MIT. For more information on which MIT objects the five POCs map to, refer to Section 7.4.1, “Relationship to the GDMO.” Each unit of persistent state is called an attribute, and an object is made up of a set of these attributes. NOTE that each POC attribute may translate to one, several, no GDMO attribute(s) in the corresponding object(s) in the MIT.

To create a new object in the MIS, first the mandatory attributes required for creation must be set either by loading values from another object or setting the values explicitly using the POC's access methods. Then, either create_with_all_attributes() or create_with_some_attributes() is called to create the object in the MIS. Note that create_with_all_attributes() only uses attributes that have been given a value. If the create method succeeds, then the POC::dn attribute will be set with the unique identifier of the new object.

To destroy an object, first the POC::dn identifier must be set, and then the destroy() method may be called to delete the object from the MIS. This is a permanent, non-reversible operation, show some care should be taken when using this method.

In order to get the attribute values of a particular object, first the POC::dn identifier must be set, then either load_all_attributes() or load_some_attributes() should be called. Once the attribute values are loaded, they stay cached within the POC and remain constant even if the values change in the MIS.

In order to set the attribute values persistently in the MIS, first the POC::dn attribute must be set, then either store_all_attributes() or store_some_attributes() may be called. Note that store_all_attributes() only stores those attributes that have been given a value.

As a point of reference, the persistence model used is a simplified version of the PMI's Image class.

Enumerations

```
enum EMObjectOperation {
    em_load,
    em_create,
    em_store,
    em_num_object_operations
};
```

Constructors, Destructor, Assignment Operator

```
    virtual ~EMObject();
```

Since this is an abstract base class, no instances of EMObject can be created.

EMObject Methods Supported By All POC Classes

```
    virtual RWBoolean exists() const;
```

Returns TRUE if the object represented by the persistent object class instance exists in the MIT. NOTE: You must have the unique identifier (EMTopoNode::dn, EMTopoType::dn, etc.) set for this method to work properly. Otherwise, FALSE will be returned.

```
    virtual EMStatus create_with_all_attributes();

    virtual EMStatus create_with_some_attributes(
        const EMIntegerSet& attributes
    );
```

These methods will create a new object in the MIS. In order for the create to succeed, the mandatory attribute required by the particular POC must be set. The new object will have its attribute values determined as follows: If create_with_all_attributes() was used, then any attribute that was given a vlue will be stored in the new object. If create_with_some_attributes() is used, then only the specified attributes are stored in the new object. In either case, any attributes which are not given a value will take on a default value defined by the GDMO for that object.

The possible error conditions are EMStatus::missing_mandatory_attribute, EMStatus::attribute_is_not_creatable, EMStatus::encode_error, and EMStatus::pmi_error.

```
    virtual EMStatus destroy();
```

This method will delete the object identified by POC::dn from the MIS. This is a permanent, non-reversible operation, show some care should be taken when using this method.

The possible error conditions are EMStatus::missing_mandatory_attribute, EMStatus::object_doesnt_exist, and EMStatus::pmi_error.

```
    virtual EMStatus load_all_attributes();
    virtual EMStatus load_some_attributes(
        const EMIntegerSet& attributes
    );
```

This method will load attributes of the object identified by POC::dn from the MIS into the POC internal cache. load_all_attributes() loads all attributes whereas load_some_attributes() only loads the specified attributes.

The possible error conditions are EMStatus::missing_mandatory_attribute, EMStatus::object_doesnt_exist, EMStatus::not_supported, and EMStatus::pmi_error.

```
    virtual EMStatus store_all_attributes();

    virtual EMStatus store_some_attributes(
        const EMIntegerSet& attributes
    );
```

This method will store attributes of the object identified by POC::dn to the MIS from the POC internal cache. store_all_attributes() stores all attributes which have been given a value whereas store_some_attributes() store only the specified attributes, without regard to whether the attributes have been give a value. Care must be taken to only specify attributes which have values, otherwise an arbitrary (usually NULL or empty) value will be stored.

The possible error conditions are EMStatus::missing_mandatory_attribute, EMStatus::object_doesnt_exist, EMStatus::attribute_is_not_storable, EMStatus::encode_error and EMStatus::pmi_error.

```
virtual void get_active_attributes(
    EMIntegerSet& set
) const;
```

This method returns the set of attributes which have been given a value.

```
virtual void clear_all_attributes();

virtual void clear_some_attributes(
    const EMIntegerSet& set
);
```

This method clear the internal memory of all or some attributes. This is useful when you want to reuse a POC instance to access a different object and don't want the previous values to remain in effect. After clear_all_attributes() is called, all attributes no longer have a value in the internal memory, including the POC::dn attribute.

Other Operators Supported by all POC classes

```
RWBoolean operator ==(
    const EMPOC&
);

RWBoolean operator !=(
    const EMPOC&
);
```

Two instance of EMPOC are considered to be equal if they each have the same attributes with a value and those values are the same for both. If one instance has a value for an attribute that the other instance doesn't have a value for, then the instances are not equal to one another. To compare only a subset of the attributes, use compare_some_attributes().

Other Methods Supported by all POC classes.

```
    RWBoolean compare_all_attributes(
        const EMPOC& other_poc
    ) const;

    RWBoolean  compare_some_attributes(
       const EMPOC& other_poc,
        const EMIntegerSet& attributes
    ) const;
```

The method compare_all_attributes() is equivalent to operator == (other_agent). The method compare_some_attributes() compares only the specified subset of attributes. For each attribute in <attributes>, either both EMPOC instances must have no value set for the attribute or if both of them have a value set for the attribute then the values must be equal. If one EMPOC instance has a value for the attribute while the other does not, then the instances are not equal to one another.

```
     RWBoolean  diff_all_attributes(
        const EMPOC& other_agent,
        EMIntegerSet& differences
    ) const;

    RWBoolean  diff_some_attributes(
        const EMPOC& other_agent,
        const EMIntegerSet& attributes,
        EMIntegerSet& differences
    ) const;
```

The diff_all_attributes() and diff_some_attributes() methods both have a return value equal to compare_all_attributes() and compare_some_attributes(), respectively, and in addition, return the set of attributes where the two instances differed if any. The EMIntegerSetIterator can then be used to iterate over <differences>.

Static Methods Supported by All POC Classes

```
    static const EMIntegerSet& get_valid_attributes(
        EMObjectOperation op = em_load
    );

    static const EMIntegerSet& get_mandatory_attributes(
        EMObjectOperation op = em_load
    );

    static const RWCString& get_attribute_name(
        EMPOC::Attribute attribute
    );
```

These static methods provide information about the EMPOC's attributes. The method get_valid_attributes() returns the set of attributes which are valid for the specified operation <op>. The method get_mandatory_attributes() returns the set of attributes which are mandatory for the specified operation <op>. If a mandatory attribute is not set when the particular operation is called an EMStatus::missing_mandatory_atribute error will result. The method returns an attribute name in string form.

7.7.6 `EMTopoNodeDn` *Class*

Inheritance: none

```
#include <topo_api/topo_api.hh>
```

Description

An instance of the `EMTopoNodeDn` class uniquely identifies one topology node out of the set of topology node objects interfaced by the EMTopoNode class.

Enumerations

```
enum NullId {
    null_id = -1
};
```

Constructors, Destructor, Assignment Operator

```
EMTopoNodeDn();
```

The default constructor creates a EMTopoNodeDn instance that is null. The `is_null()` method returns true for this instance.

```
    EMTopoNodeDn(
            const RWCString& system_name,
            long unique_id
    );
```

Creates an EMTopoNodeDn instance that is uniquely identified by <*system_name*> and <*unique_id*>.

Operators

```
    RWBoolean operator ==(
        const EMTopoNodeDn& dn
    ) const;
    RWBoolean operator !=(
        const EMTopoNodeDn& dn
    ) const;
```

Two instances are equal if they have both the same system name and the same unique name or if they are both null.

Access Methods

```
    const RWCString& system_name() const;
    void system_name(
           const RWCString& system_name
    );
```

The name of the MIS where the topology node is stored.

```
    long unique_id() const;
    void unique_id(
           long unique_id
    );
```

The topology node identifier. This identifier is unique within a single MIS.

General Methods

```
    void make_null();
    RWBoolean is_null() const;
```

Sets to null value and tests for null value. A null value means that the EMTopoNodeDn does not refer to any topology node.

Related Global Operators

```
ostream& operator<<(ostream& s, const EMTopoNodeDn& dn);
```

The stream output operator << is defined to provide an easy was to print out the value of EMTopoNodeDn.

7.7.6.1 EMTopoNode *Class*

Inheritance: EMObject

```
#include <topo_api/topo_api.hh>
```

Description

The EMTopoNode class....

In Table 7-4, the attribute key is:

C=Attribute can be set at creation time.
S=Attribute can be set after creation time.
M=Mandatory; attribute must be set for operation to succeed.
X=Allowed; attribute can be set as an option.

Table 7-4 EMTopoNodes Attributes Table

Attribute Enum	C	S	Description
dn		M	Unique identifier
name	M	X	Name of this node (need not be unique)
topology_pathnames			List of all topology pathnames for the node. At a minimum, there will be one pathname for each parent. However, since each parent can also have more than one parent, and so on, the actual number of pathnames may be higher. Example list: "/Root/Internet/129.146.74.0/host-45", "/Root/hosts/host-45"
type_name	M	X	Type name of this node
managed_objects	X	X	List of DNs (in ASCII slash format) of the managed objects in the MIT associated with this node
cmip_agents	X	X	list of cmip agents which have managed objects listed as part of managed_objects attribute.
snmp_agents	X	X	list of snmp agents which have managed objects listed as part of managed_objects attribute.
rpc_agents	X	X	list of rpc agents which have managed objects listed as part of managed_objects attribute.
parents	M	X	List of topology nodes that contain this node in the topology directed acyclic graph (DAG)

Table 7-4 EMTopoNodes Attributes Table

Attribute Enum	**C**	**S**	**Description**
`children`			List of topology nodes that are contained by this node
`view_children`			Subset of children whose type_name is a view; that is, EMTopoType::is_view(type_name) returns TRUE.
`links`	X	X	List of Link topology nodes connected to this node
`propagate_peers`	X	X	List of topology nodes for this node's severity propagation if `is_severity_attribute` is true
`is_severity_propagated`	X	X	If TRUE, then the node's severity will be propagated to each of its parents where it will factor it the calculation of their severity.
`state`	X	X	Can be used to store an integer value
`severity`	X	X	If the Alarm Service is running, the severity indicates the highest severity alarm posted against any of the `managed_objects`. Note: Normally, an application never sets the severity attribute; this attribute is automatically updated from the Alarm Service.
`propagated_severity`			If the Alarm Service is running, the propagated_severity indicates the highest severity among this node's severity and the severity of any children of this node who have their `is_severity_propagated` flag set to true.
`display_statuses`	X	X	A user-defined list of tags, value pairs, such as { { "CPUUsage", 45 } , { "DiskLoad" , 2345 } }.
`geographical_location`	X	X	The latitude and longitude in degrees floating-point of the location of this node.
`layer_name`	X	X	The layer that this node belongs to.
`user_data`	X	X	User-defined data that should contain values for each attribute name listed in EMTopoType::user_data_attribute_names for the <*type_name*> of this node.
`logical_locations`	X	X	A list of locations where the node appears in each of its parent views.
`view_background_image_filename`	X	X	Absolute pathname of Sun raster file image to be displayed when the viewer canvas is in logical view mode.
`view_map_config_filename`	X	X	Absolute pathname of geographical map configuration (GMC) file to be displayed when the Viewer canvas is in geographical view mode.

Table 7-4 `EMTopoNodes` Attributes Table

Attribute Enum	**C**	**S**	**Description**
`view_default_geo_area`	X	X	Default geographical area (specified as a center and view width in km) to be displayed when the view_map_config_filename is first displayed.
`monitor_rotation`	X	X	Number of degrees to rotate the monitor node.
`monitor_visible_children`		X	Subset of the children list of nodes that should appear in the monitor sections.
`monitor_hidden_children`			List of children nodes remaining when `monitor_visible_children` list of nodes are subtracted from the children list of nodes. These nodes do not appear in a monitor section, even if there are empty sections.
`monitor_max_visible_children`			Maximum number of visible children supported by the particular type of monitor.

Example

```
#include <topo_api/topo_api.hh>

// this assumes the snmp_agent was created
// elsewhere
RWBoolean
create_host(
    const EMTopoNodeDn& parent_dn,
    const EMSnmpAgentDn& snmp_agent_dn
)
{
    EMTopoNode node;

    node.set_name(snmp_agent_dn.unique_name());
    node.set_type_name(EMTopoTypeDn::host);
    node.add_parent(parent_dn);
    // for hosts, we use the DN of the snmp agent
    // object as the managed object.
    node.add_managed_object(snmp_agent_dn.slash_form());

    EMStatus status;
    if (!(status = node.create_with_all_attributes())) {
            cerr << "Error: " << status << endl;
            return FALSE;
    }
    return TRUE;
}
```

Enumerations

```
enum EMTopoNode::AttributeType {
    all_attributes,
    common_attributes,
    view_only_attributes,
    monitor_only_attributes,
    link_only_attributes,
    device_only_attributes,
    num_attribute_types
};

enum EMTopoNode::Attribute {
     dn=0,
     name,
     type_name,
     managed_objects,
     cmip_agent,
     snmp_agent,
     rpc_agent,
     parents,
     children,
     children_containers_only,
     links,
     propagate_peers,
     is_severity_propagated,
     state,
     severity,
     propagated_severity,
     display_statuses,
     geographical_location,
     layer_name,
    user_data,
    logical_locations,
    view_background_image_filename,
     view_map_config_filename,
     view_default_geo_area,
    monitor_rotation,
     monitor_visible_children,
     monitor_hidden_children,
     monitor_max_visible_children,
     num_attributes
};
```

```
 enum EMTopoNode::Severity {
   indeterminate = 0,
   critical = 1,
   major = 2,
   minor = 3,
   warning = 4,
   cleared = 5,
   min_severity = indeterminate,
   max_severity = cleared,
   num_severity = max_severity - min_severity + 1
};
```

Structs

```
struct EMTopoNode::Location
{
    long x;
    long y;
    long z;

    Location();
    Location(long p_x,long p_y,long p_z=0);
    RWBoolean operator == (const Location& l) const;
    RWBoolean operator != (const Location& l) const;

    ostream& operator<<(ostream& s, const Location& l);
};
```

```
struct EMTopoNode::LocationInParent
{
    EMTopoNodeDn parent;
    Location location;

    LocationInParent();

    LocationInParent(
        const EMTopoNodeDn& p_parent,
        const Location& p_location
    );

    RWBoolean operator == (
        const LocationInParent& l
    ) const;
    RWBoolean operator != (
        const LocationInParent& l
    ) const;

    ostream& operator<<(
        ostream& s,
        const LocationInParent& l
    );
};
```

```
struct EMTopoNode::GeoLocation
{
    double longitude;
    double latitude;

    GeoLocation();
    GeoLocation(double p_longitude,double p_latitude);
    RWBoolean operator == (const GeoLocation& l) const;
    RWBoolean operator != (const GeoLocation& l) const;

    ostream& operator<<(ostream& s, const GeoLocation& l);
};
```

```
struct EMTopoNode::DisplayStatus
{
    RWCString label;
    long value;

    DisplayStatus();
    DisplayStatus(const RWCString& p_label,long p_value);
    RWBoolean operator == (const DisplayStatus& d) const;
    RWBoolean operator != (const DisplayStatus& d) const;

    ostream& operator<<(ostream& s, const DisplayStatus& d);
};
```

```
struct EMTopoNode::UserDatum
{
    RWCString attribute_name;
    Morf value;

    UserDatum();
    UserDatum(
        const RWCString& p_attribute_name,
        const Morf& p_value);
    RWBoolean operator == (
        const UserDatum& d
    ) const;
    RWBoolean operator != (
        const UserDatum& d) const;
    ostream& operator<<(ostream& s,
        const UserDatum& d);
};
```

Constructors, Destructor, Assignment Operator

```
     EMTopoNode();
     EMTopoNode(
         const EMTopoNodeDn& id
     );

     EMTopoNode(
         const EMTopoNode& node
     );
     ~EMTopoNode();

     EMTopoNode& operator =(
         const EMTopoNode&
     );
```

Access Methods

```
     EMStatus get_dn(
         EMTopoNodeDn& dn
     ) const;

     EMStatus set_dn(
         const EMTopoNodeDn& dn
     );
```

```
     EMStatus get_name(
         RWCString& name
     ) const;

     EMStatus set_name(
         const RWCString& name
     );
```

```
    EMStatus get_topology_pathnames(
        RWTValSlist<RWCString>& topology_pathnames
    ) const;
```

```
    EMStatus get_type_name(
        RWCString& type_name
    ) const;

    EMStatus set_type_name(
        const RWCString& type_name
    );
```

```
    EMStatus get_severity(
        EMTopoNode::Severity& severity
    ) const;

    EMStatus set_severity(
        EMTopoNode::Severity severity
    );

    EMStatus get_propagated_severity(
        EMTopoNode::Severity& severity
    ) const;
```

```
    EMStatus get_is_severity_propagated(
        RWBoolean& is_severity_propagated
    ) const;
    EMStatus set_is_severity_propagated(
        const RWBoolean& is_severity_propagated
    );
```

```
EMStatus get_propagate_peers(
    RWTValSlist<EMTopoNodeDn>& peers
) const;

EMStatus set_propagate_peers(
    const RWTValSlist<EMTopoNodeDn>& peers
);

EMStatus add_propagate_peer(
    const EMTopoNodeDn& peer
);

EMStatus remove_propagate_peer(
    const EMTopoNodeDn& peer
);
```

```
EMStatus get_state(
    long& state
) const;

EMStatus set_state(
    long state
);
```

```
EMStatus get_display_statuses(
    RWTValSlist<EMTopoNode::DisplayStatus>& statuses
) const;

EMStatus set_display_statuses(
    const RWTValSlist<EMTopoNode::DisplayStatus>& statuses
);

EMStatus get_display_status(
    const RWCString& label,
    long& status
) const;

EMStatus add_display_status(
    const RWCString& label,
    long status
);

EMStatus remove_display_status(
    const RWCString& label
);
```

```
EMStatus get_children(
    RWTValSlist<EMTopoNodeDn>& children
) const;

EMStatus get_children_containers_only(
    RWTValSlist<EMTopoNodeDn>& children
) const;
```

```
EMStatus get_parents(
    RWTValSlist<EMTopoNodeDn>& parents
) const;

EMStatus set_parents(
    const RWTValSlist<EMTopoNodeDn>& parents
);

EMStatus add_parent(
    const EMTopoNodeDn& parent
);

EMStatus remove_parent(
    const EMTopoNodeDn& parent
);
```

```
EMStatus get_links(
    RWTValSlist<EMTopoNodeDn>& links
) const;

EMStatus set_links(
    const RWTValSlist<EMTopoNodeDn>& links
);

EMStatus add_link(
    const EMTopoNodeDn& link
);

EMStatus remove_link(
    const EMTopoNodeDn& link
);
```

```
    EMStatus get_logical_location(
        const EMTopoNodeDn& parent,
        EMTopoNode::Location& location
    ) const;

    EMStatus set_logical_location(
        const EMTopoNodeDn& parent,
        const EMTopoNode::Location& location
    );

    EMStatus get_logical_locations(
        RWTValSlist<EMTopoNode::LocationInParent>& locations
    ) const;

    EMStatus set_logical_locations(
     const RWTValSlist<EMTopoNode::LocationInParent>& locations
    );
```

```
    EMStatus get_geographical_location(
        EMTopoNode::GeoLocation& location,
        RWBoolean& is_null
    ) const;

    EMStatus set_geographical_location(
        const EMTopoNode::GeoLocation& location,
        RWBoolean is_null = FALSE
    );
```

```
    EMStatus get_layer_name(
        RWCString& name
    ) const;

    EMStatus set_layer_name(
        const RWCString& name
    );
```

```
EMStatus get_managed_objects(
    RWTValSlist<RWCString>& managed_objects
) const;

EMStatus set_managed_objects(
    const RWTValSlist<RWCString>& managed_objects
);

EMStatus add_managed_object(
    const RWCString& managed_object
);

EMStatus remove_managed_object(
    const RWCString& managed_object
);
```

```
EMStatus get_cmip_agents(
    RWTValSlist<EMCmipAgentDn>& agents
) const;
```

```
EMStatus get_snmp_agents(
    RWTValSlist<EMSnmpAgentDn>& agents
) const;
```

```
EMStatus get_rpc_agents(
    RWTValSlist<EMRpcAgentDn>& agents
) const;
```

```
EMStatus get_user_data(
    RWTValSlist<EMTopoNode::UserDatum>& user_data
) const;

EMStatus set_user_data(
    const RWTValSlist<EMTopoNode::UserDatum>& user_data
);

EMStatus get_user_datum(
    const RWCString& attribute_name,Morf& morf
) const;

EMStatus add_user_datum(
    const RWCString& attribute_name,const Morf& morf
);

EMStatus remove_user_datum(
    const RWCString& attribute_name
);
```

```
EMStatus get_view_background_image_filename(
    RWCString& filename
) const;

EMStatus set_view_background_image_filename(
    const RWCString& filename
);
```

```
EMStatus get_view_map_config_filename(
    RWCString& filename
) const;

EMStatus set_view_map_config_filename(
    const RWCString& filename
);
```

```
EMStatus get_view_default_geo_area(
    EMTopoNode::GeoLocation& center,
    double& width_in_km,
    RWBoolean& is_null
) const;

EMStatus set_view_default_geo_area(
    const EMTopoNode::GeoLocation& center,
    double width_in_km,
    RWBoolean is_null = FALSE
);
```

```
EMStatus get_monitor_rotation(
    long& rotation,
    RWBoolean& is_null
) const;

EMStatus set_monitor_rotation(
    long rotation,
    RWBoolean is_null = FALSE
);
```

```
EMStatus get_monitor_visible_children(
    RWTValSlist<EMTopoNodeDn>& children
) const;

EMStatus set_monitor_visible_children(
    const RWTValSlist<EMTopoNodeDn>& children
);

EMStatus add_monitor_visible_child(
    const EMTopoNodeDn& child
);

EMStatus remove_monitor_visible_child(
    const EMTopoNodeDn& child
);
```

```
EMStatus get_monitor_hidden_children(
    RWTValSlist<EMTopoNodeDn>& children
) const;
```

```
EMStatus get_monitor_max_visible_children(
    long& max_children
) const;
```

Static Methods for Event Subscription

The EMTopoNode class provides an event subscription service so that clients can be notified when a topology node is created, deleted, or modified.

```
struct EMTopoNodeCallbackData
{
    EMEventType event_type;
    EMTopoNodeDn node_dn;
    EMTopoNode changes;
    void* client_data;
};

typedef void (*Callback)(
    const EMTopoNodeCallbackData& cbd
);
```

To register for events, the client must provide a callback function with of type EMTopoNode::Callback. When the client's callback is called, the <cbd> parameter will be filled in with information about the event. The <event_type> field indicates the type of event: em_create_event, em_delete_event, or em_change_event. The <node_dn> parameter uniquely identifies the topology node that was created, deleted, or modified. For em_change_event only, the <changes> parameter will contain the new values for all attributes which changed. To get a list of the changed attributes, call EMTopoNode::get_active_attributes(). The normal EMTopoNode access methods may be used to get the new attribute values. Finally, the <client_data> field is the same as the <client_data> parameter of EMTopoNode::register_callback(). An EMTopoNode events example is $EM_HOME/src/topo_api/topo_events.cc.

```
    void register_callback(
            EMEventType event,
            Callback callback,
            void* client_data
    );
```

Registers <callback> to be called when <event> occurs on any topology node. If <event> equals em_any_event, then <callback> will be called for any of em_create_event, em_delete_event, or em_change_event. The parameter <client_data> will be used to initiaze the <client_data> field in the EMTopoNodeCallbackData struct. Note: If the same <callback> has already been registered for the same <event>, then the callback will not be added a second time; However, the <client_data> will replace the previous <client_data>.

```
    void unregister_callback(
           EMEventType event,
           EMTopoNode::Callback callback
    );
```

Removes <callback> that was previously registered for <event> events. Note if <callback> was registered multiple times with different <event> parameters, the <callback> will only be removed for this <event>.

Related Global Operators

```
    ostream& operator<<(
           ostream& s,
           const EMTopoNode& node
       );
```

The stream output operator << is defined to provide an easy was to print out the attribute values of EMTopoNode.

7.7.7 EMTopoTypeDn *Class*

Inheritance: none

```
    #include <topo_api/topo_api.hh>
```

Description

An instance of the EMTopoTypeDn class uniquely identifies one topology type out of the set of topology types interfaced by the EMTopoType.

Constants

```
    static const RWCString
          EMTopoTypeDn::container,
          EMTopoTypeDn::device,
          EMTopoTypeDn::link,
          EMTopoTypeDn::monitor;
```

Convenience constants for the default base types.

Constructors, Destructor, Assignment Operator

```
   EMTopoTypeDn();
   EMTopoTypeDn(
       const RWCString& system_name,
       const RWCString& unique_name
   );
```

The default constructor creates a null object.The other constructor takes the MIS name where the object is stored and the topology type name.

Operators

```
   RWBoolean operator ==(
       const EMTopoTypeDn& dn
   ) const;

   RWBoolean operator !=(
       const EMTopoTypeDn& dn
   ) const;
```

Two instances are equal if they have both the same system name and the same unique name or if they are both null.

Access Methods

```
const RWCString& system_name() const;

void system_name(
    const RWCString& system_name
);
```

The name of the MIS where the topology type is stored.

```
const RWCString& unique_name() const;

void unique_name(
    const RWCString& unique_name
);
```

The name of the topology type. This name is unique within a single MIS.

General Methods

```
void make_null();
RWBoolean is_null() const;
```

Sets to null value and tests for null value. A null value means that the EMTopoTypeDn does not refer to any topology type.

Related Global Operators

```
    ostream& operator<<(
            ostream& s,
            const EMTopoTypeDn& dn
    );
```

7.7.8 EMTopoType *Class*

Inheritance: EMObject

```
    #include <topo_api/topo_api.hh>
```

Description

An instance of the EMTopoType class represents a topology type. Every topology node is classified as a particular topology type. The topology types form a hiearchy with the four base types “Container”, “Device”, “Monitor”, and “Link” with other subtypes derived from them. Beyond the standard POC methods which allow you to create, delete, compare, etc. topology types, the EMTopoType class provides the following additional services:

- static methods is_container(), is_device(), is_monitor(), is_link(), and is_view() (equivalent to is_monitor() or is_container()) can be used to categorize topology types.

The EM topo type attributes are described in Table 7-5.

The attribute key is:

C=Attribute can be set at creation time.
S=Attribute can be set after creation time.
M=Mandatory; attribute must be set for operation to succeed.
X=Allowed; attribute can be set as an option.

Table 7-5 EM TopoType Attributes

Attribute Enum	**C**	**S**	**Description**
dn	M	M	Unique identifier
base_type	M		The parent topology type of this type.
all_base_types			All ancestors of this type.
sub_types			All topology types contained by this type.
legal_children	X	X	List of legal topology types of topology nodes that can be contained by a topology node of this type within the topology hierarchy.
layer_name	M	X	Name of the layer that includes topology nodes of this type.
user_data_attribute_names	X	X	A list of GDMO attribute names that define the contents of the EMTopoNode::user_data attribute for EMTopoNodes of this type.

Example

```
RWBoolean
create_topo_type(
    const RWCString& system_name,
    const RWCString& type_name
)
{
    EMTopoType type(EMTopoTypeDn(system_name,type_name));

    type.set_base_type(EMTopoTypeDn::device);
    type.set_layer_name(type_name);

    EMStatus status;
    if (!(status = type.create_with_all_attributes())) {
           cerr << “Error: “ << status << endl;
           return FALSE;
    }
    return TRUE;
}
```

Enumerations

```
    enum EMTopoType::Attribute {
        dn=0,
        base_type,
        sub_types,
        legal_children,
        layer_name,
        user_data_attribute_names,
        num_attributes
    };
```

Constructors, Destructor, Assignment Operator

```
EMTopoType(
    const EMTopoTypeDn& dn
);

EMTopoType(
    const EMTopoType& topo_type
);

~EMTopoType();
EMTopoType& operator =(
    const EMTopoType& topo_type
);
```

Access Methods

```
EMStatus get_dn(
    EMTopoTypeDn& dn
) const;

EMStatus set_dn(
    const EMTopoTypeDn& dn
);
```

```
EMStatus get_base_type(
    RWCString& type_name
) const;

EMStatus set_base_type(
   const RWCString& type_name
);

EMStatus get_all_base_types(
       RWTValSlist<RWCString>& base_types
) const;
```

```
EMStatus get_sub_types(
    RWTValSlist<RWCString>& sub_types
) const;
```

```
EMStatus get_legal_children(
    RWTValSlist<RWCString>& children
) const;

EMStatus add_legal_child(
    const RWCString& child
);
```

```
EMStatus get_layer_name(
    RWCString& layer_name
) const;

EMStatus set_layer_name(
    const RWCString& layer_name
);
```

```
EMStatus get_user_data_attribute_names(
    RWTValSlist<RWCString>& names
) const;

EMStatus set_user_data_attribute_names(
    const RWTValSlist<RWCString>& names
);

EMStatus add_user_data_attribute_name(
    const RWCString& name
);

EMStatus remove_user_data_attribute_name(
    const RWCString& name
);
```

Static Methods

```
    static RWBoolean is_container(
        const RWCString& type_name
    );

    static RWBoolean is_monitor(
        const RWCString& type_name
    );

    static RWBoolean is_view(
        const RWCString& type_name
    );

    static RWBoolean is_device(
        const RWCString& type_name
    );

    static RWBoolean is_link(
        const RWCString& type_name
    );
```

These methods return TRUE if <type_name> is a subtype of the indicated base type. The method is_view() is special because there is no base type named 'View'; is_view() is equivalent to is_container() or is_monitor().

Global Operators

```
    ostream& operator<<(
        ostream& s,
        const EMTopoType& type
    );
```

The stream output operator << is defined to provide an easy was to print out the attribute values of EMTopoType.

7.7.9 EMAgent *Class*

Inheritance: EMObject

```
#include <topo_api/topo_api.hh>
```

Description

The EMAgent class is an abstract class that contains the agent interface common between EMCmipAgent, EMRpcAgent, and EMSnmpAgent.

Enumerations

```
    qenum EMAgent::Attribute {
           operational_state,
           administrative_state,
           num_attributes
    };
```

```
    enum EMAgent::AdministrativeState {
        locked,
        unlocked,
        shuttingdown,
        num_administrative_states
    };
```

Used to suspend and resume the proxy activity relative to the Internet Agent. The EMAgent::unlocked state means that the proxy must continue to perform, or resume performing, proxy activities on

Class Reference: EMAgent Class

behalf of the Internet agent. The EMAgent::locked state means that the proxy must not perform, or suspend performing, proxy activities on behalf of the Internet agent.

```
    enum EMAgent::OperationalState {
        disabled,
        enabled,
        num_operational_states
    };
```

Indicates the perceived state of the Internet agent. The EMAgent::enabled state means that the Internet agent is operational, as perceived by the proxy: it can be reached. The EMAgent::disabled state means that the Internet agent is not operational, as perceived by the proxy; it cannot be reached.

Access Methods

```
    EMStatus get_operational_state(
        OperationalState& operational_state
    ) const;
```

Note: The operational_state is read-only.

```
    EMStatus get_administrative_state(
        AdministrativeState& administrative_state
    ) const;

    EMStatus set_administrative_state(
        const AdministrativeState& administrative_state
    );
```

7.7.10 `EMCmipAgentDn` *Class*

Inheritance: none

```
    #include <topo_api/topo_api.hh>
```

Description

An instance of the `EMCmipAgentDn` class uniquely identifies one rpc agent object out of the set of rpc agent objects interfaced by the EMCmipAgent persistent object class.

Constructors, Assignment Operators

```
EMCmipAgentDn();

EMCmipAgentDn(
    const RWCString& system_name,
    const RWCString& unique_name
);
```

The default constructor creates a null object. The other constructor takes the MIS name where the object is stored and the cmip agent name.

Operators

```
RWBoolean operator ==(
    const EMCmipAgentDn& dn
) const;

RWBoolean operator !=(
    const EMCmipAgentDn& dn
) const;
```

Two instances are equal if they have both the same system name and the same unique name or if they are both null.

Access Methods

```
const RWCString& system_name() const;

void system_name(
    const RWCString& system_name
);
```

The name of the MIS where the cmip agent object is stored.

```
const RWCString& unique_name() const;

void unique_name(
    const RWCString& unique_name
);
```

The name of the cmip agent object. This name is unique within a single MIS.

General Methods

```
void make_null();
RWBoolean is_null() const;
```

Sets to null value and tests for null value. A null value means that the EMCmipAgentDn does not refer to any cmip agent object.

Global Operators

```
ostream& operator<<(ostream& s, const EMCmipAgentDn& dn);
```

The stream output operator << is defined to provide an easy was to print out the value of EMCmipAgentDn.

7.7.11 EMCmipAgent Class

Inheritance: EMAgent <- EMObject

```
    #include <topo_api/topo_api.hh>
```

Description

An instance of the EMCmipAgent class represents the MIS object which contains configuration information for an CMIP agent. The configuration information includes the CMIP MPA hostname and port number, list of managed objects DNs, network SAP, transport selector, presentation selector, session selector, and application entity title (AET). NOTE: This class does not provide an interface to the agent's managed objects, but only to Solstice EM's configuration information for the agent.

Table 7-6 gives the CmipAgent attributes.

The attribute key is:

C=Attribute can be set at creation time.
S=Attribute can be set after creation time.
M=Mandatory; attribute must be set for operation to succeed.
X=Allowed; attribute can be set as an option.

Table 7-6 EMCmipAgent Attributes

Attribute Enum	**C**	**S**	**Description**
`dn`	M	M	Unique identifier
`operational_state`			EMAgent::disabled or enabled
`administrative_state`	M	X	EMAgent::locked, unlocked, or shuttingdown
`mpa_address_info`	X	X	mpa hostname and port number
`agent_address_tag`	X	X	defines format of agent_address_info
`agent_address_info`	M	X	agent address information in format defined by agent_address_tag
`managed_objects`	M	X	list of DNs in slash format of managed objects located on agent. Note that the multiple cmip agent configurations can be created for the same cmip mpa but with a different set of managed objects for each.
`application_entity_title`	M	X	application entity title (AET)

Table 7-6 EMCmipAgent Attributes

Attribute Enum	**C**	**S**	**Description**
presentation_selector	M	X	OSI presentation selector
session_selector	M	X	OSI session selector
transport_selector	M	X	OSI transport selector
network_sap	M	X	OSI network sap

Example

```
setup_mis_mis_connection(
    const RWCString& manager_hostname,
    const RWCString& agent_hostname
)
{
    EMCmipAgent cmip_agent(EMCmipAgentDn(manager_hostname,
                                         agent_hostname));
    cmip_agent.set_administrative_state(EMAgent::unlocked);
    cmip_agent.set_mpa_address_info(agent_hostname,5555);

    RWCString managed_object("/systemId=name:\"");
    managed_object.append(agent_hostname).append("\"");
    cmip_agent.add_managed_object(managed_object);
    cmip_agent.set_application_entity_title("objectIdentifier :
{ 1 2 3 4 }");
    cmip_agent.set_presentation_selector("");
    cmip_agent.set_session_selector("");
    cmip_agent.set_transport_selector("");
    char buffer[128];
    sprintf(buffer,"%s:%d",agent_hostname,5555);
    cmip_agent.set_network_sap(buffer);
    // String: {psel,ssel,tsel,nsap}
    cmip_agent.set_agent_address_tag(8);
    sprintf(buffer,"{,,,%s:%d",agent_hostname,5555);
    cmip_agent.set_agent_address_info(buffer);

    EMStatus status;
    if (!(status = cmip_agent.create_with_all_attributes())) {
        cerr << "Error: " << status << endl;
    }
}
```

Enumerations

```
enum EMCmipAgent::Attribute {
    dn = EMAgent::num_attributes,
    mpa_address_info,
    agent_address_info,
    agent_address_tag,
    managed_objects,
    application_entity_title,
    presentation_selector,
    session_selector,
    transport_selector,
    network_sap,
    num_attributes
};
```

Access Methods

```
EMStatus get_dn(
     EMCmipAgentDn& dn
) const;

EMStatus set_dn(
     const EMCmipAgentDn& dn
);
```

```
EMStatus get_mpa_address_info(
    RWCString& hostname,
    int& port_number,
    RWBoolean& is_null
) const;

EMStatus set_mpa_address_info(
    const RWCString& hostname,
    int port_number,
    RWBoolean is_null = FALSE
);
```

```
EMStatus get_managed_objects(
    RWTValSlist<RWCString>& dns
) const;

EMStatus set_managed_objects(
    const RWTValSlist<RWCString>& dns
);

EMStatus add_managed_object(
    const RWCString& dn
);

EMStatus remove_managed_object(
    const RWCString& dn
);
```

```
EMStatus get_network_sap(
    RWCString& network_sap
) const;

EMStatus set_network_sap(
    const RWCString& network_sap
);
```

```
EMStatus get_agent_address_info(
    RWCString& agent_address_info
) const;

EMStatus set_agent_address_info(
    const RWCString& agent_address_info
);
```

```
EMStatus get_agent_address_tag(
    int& agent_address_tag
) const;

EMStatus set_agent_address_tag(
    int agent_address_tag
);
```

```
EMStatus get_presentation_selector(
    RWCString& presentation_selector
) const;

EMStatus set_presentation_selector(
    const RWCString& presentation_selector
);
```

```
EMStatus get_session_selector(
    RWCString& session_selector
) const;

EMStatus set_session_selector(
    const RWCString& session_selector
);
```

```
EMStatus get_transport_selector(
    RWCString& transport_selector
) const;

EMStatus set_transport_selector(
    const RWCString& transport_selector
);
```

```
EMStatus get_application_entity_title(
    RWCString& application_entity_title
) const;

EMStatus set_application_entity_title(
    const RWCString& application_entity_title
);
```

Global Operators

```
ostream& operator<<(ostream& s, const EMCmipAgent& agent);
```

The stream output operator << is defined to provide an easy was to print out the attribute values of EMCmipAgent.

7.7.12 `EMRpcAgentDn` *Class*

Inheritance: none

```
#include <topo_api/topo_api.hh>
```

Description

An instance of the `EMRpcAgentDn` class uniquely identifies one rpc agent object out of the set of rpc agent objects interfaced by the EMRpcAgent persistent object class.

Constructors, Assignment Operators

```
    EMRpcAgentDn();

    EMRpcAgentDn(
        const RWCString& system_name,
        const RWCString& unique_name
    );
```

The default constructor creates a null object.The other constructor takes the MIS name where the object is stored and the rpc agent name.

Operators

```
    RWBoolean operator ==(
        const EMRpcAgentDn& dn
    ) const;

    RWBoolean operator !=(
        const EMRpcAgentDn& dn
    ) const;
```

Two instances are equal if they have both the same system name and the same unique name or if they are both null.

Access Methods

```
const RWCString& system_name() const;

void system_name(
    const RWCString& system_name
);
```

The name of the MIS where the rpc agent object is stored.

```
const RWCString& unique_name() const;

void unique_name(
    const RWCString& unique_name
);
```

The name of the rpc agent object. This name is unique within a single MIS.

General Methods

```
void make_null();
RWBoolean is_null() const;
```

Sets to null value and tests for null value. A null value means that the EMRpcAgentDn does not refer to any rpc agent object.

Global Operators

```
ostream& operator<<(ostream& s, const EMRpcAgentDn& dn);
```

The stream output operator << is defined to provide an easy was to print out the value of EMRpcAgentDn.

7.7.13 `EMRpcAgent` *Class*

Inheritance: EMAgent <- EMObject

```
    #include <topo_api/topo_api.hh>
```

Description

An instance of the EMRpcAgent class represents the MIS object which contains configuration information for an RPC agent. The configuration information includes the read and write community strings, and supported schemas. NOTE: This class does not provide an interface to the agent's managed objects, but only to Solstice EM's configuration information for the agent.

Table 7-7 gives the `EMRpcAgent` attributes.

The attribute key is:

C=Attribute can be set at creation time.
S=Attribute can be set after creation time.
M=Mandatory; attribute must be set for operation to succeed.
X=Allowed; attribute can be set as an option.

Table 7-7 EMRpcAgent Attributes

Attribute Enum	**C**	**S**	**Description**
`dn`	M	M	Unique identifier
`operational_state`			EMAgent::disabled or enabled
`administrative_state`	M	X	EMAgent::locked, unlocked, or shuttingdown
`get_community_string`	X	X	e.g. “public”, “private”
`set_community_string`	X	X	e.g. “public”, “private”
`schemas`	M	X	list of rpc_proxy_hostname and rpc_name pairs, e.g. “ultra-server”, “RPC Proxy -ping”

Example

```
RWBoolean
create_rpc_agent(
    const RWCString& system_name,
    const RWCString& rpc_agent_name
)
{
    EMRpcAgent rpc_agent(EMRpcAgentDn(system_name,
                      rpc_agent_name));
    rpc_agent.set_administrative_state(EMAgent::unlocked);
    EMRpcAgent::Schema schema(“proxy-hostname”,
                      ”RPC Proxy -ping”);
    rpc_agent.add_schema(schema);

    EMStatus status;
    if (!(status = rpc_agent.create_with_all_attributes())) {
           cerr << “Error: “ << status << endl;
           return FALSE;
    }
    return TRUE;
}
```

Enumerations

```
    enum EMRpcAgent::Attribute {
        dn = EMAgent::num_attributes,
        get_community_string,
        set_community_string,
        schemas,
        num_attributes
    };
```

These are the attributes specific to EMRpcAgent, in addition to the attribute defined in EMAgent which are common to EMCmipAgent, EMRpcAgent, and EMSnmpAgent.

Structs

```
struct EMRpcAgent::Schema
{
    RWCString name;
    RWCString proxy_hostname;

    Schema();

    Schema(const RWCString& p_name,
           const RWCString& p_proxy_hostname);

   RWBoolean operator ==(const Schema& schema) const;

   RWBoolean operator !=(const Schema& schema) const;
}

ostream& operator<<(ostream& s, const Schema& schema);
```

The struct EMRpcAgent::Schema is used to store a rpc proxy hostname and rpc method pairing. Each EMRpcAgent can be configured to support any number of schemas.

Constructors, Destructor, Assignment Operator

```
EMRpcAgent(
    const EMRpcAgentDn& rpc_agent_dn
);

EMRpcAgent(
    const EMRpcAgent& rpc_agent
);

~EMRpcAgent();

EMRpcAgent& operator =(
    const EMRpcAgent&
);
```

Access Methods

```
EMStatus get_dn(
    EMRpcAgentDn& dn
) const;

EMStatus set_dn(
    const EMRpcAgentDn& dn
);
```

```
EMStatus get_get_community_string(
    RWCString& get_communitry_string
) const;

EMStatus set_get_community_string(
    const RWCString& get_community_string
);

EMStatus get_set_community_string(
    RWCString& set_community_string
) const;

EMStatus set_set_community_string(
    const RWCString& set_community_string
);
```

```
    EMStatus get_schemas(
        RWTValSlist<EMRpcAgent::Schema>& schemas
    ) const;

    EMStatus set_schemas(
        const RWTValSlist<EMRpcAgent::Schema>& schemas
    );

    EMStatus add_schema(
        const EMRpcAgent::Schema& schema
    );

    EMStatus remove_schema(
        const EMRpcAgent::Schema& schema
    );
```

Global Operators

```
    ostream& operator<<(
        ostream& s,
        const EMRpcAgent& agent
    );
```

The stream output operator << is defined to provide an easy was to print out the attribute values of EMRpcAgent.

7.7.14 `EMSnmpAgentDn` *Class*

Inheritance: none

```
    #include <topo_api/topo_api.hh>
```

Description

An instance of the EMSnmpAgentDn class uniquely identifies one snmp agent object out of the set of snmp agent objects interfaced by the EMSnmpAgent persistent object class.

Constructors, Destructor, Assignment Operator

```
EMSnmpAgentDn();

EMSnmpAgentDn(
    const RWCString& system_name,
    const RWCString& unique_name
);
```

The default constructor creates a null object.The other constructor takes the MIS name where the object is stored and the snmp agent name.

Operators

```
RWBoolean operator ==(
    const EMSnmpAgentDn& dn
) const;

RWBoolean operator !=(
    const EMSnmpAgentDn& dn
) const;
```

Two instances are equal if they have both the same system name and the same unique name or if they are both null.

Access Methods

```
const RWCString& system_name() const;
void system_name(
    const RWCString& system_name
);
```

The name of the MIS where the snmp agent object is stored.

```
const RWCString& unique_name() const;

void unique_name(
    const RWCString& unique_name
);
```

The name of the snmp agent object which is unique on one MIS. Combined with the system_name, the pair form a globally unique identifier.

General Methods

```
void make_null();

RWBoolean is_null() const;
```

Sets to null value and tests for null value. A null value means that the EMSnmpAgentDn does not refer to any snmpagent object.

Global Operators

```
ostream& operator<<(
    ostream& s,
    const EMSnmpAgentDn& dn
);
```

`The stream output operator << is defined to provide an easy was to print out the value of EMSnmpAgentDn.`

7.7.15 `EMSnmpAgent` *Class*

Inheritance: EMAgent <- EMObject

```
#include <topo_api/topo_api.hh>
```

Description

Table 7-8 gives the SnmpAgent attributes.

The attribute key is:

C=Attribute can be set at creation time.
S=Attribute can be set after creation time.
M=Mandatory; attribute must be set for operation to succeed.
X=Allowed; attribute can be set as an option.

Table 7-8 `EMSnmpAgent` Attributes

Attribute Enum	C	S	Description
`dn`	M	M	Unique identifier, which includes the administrative name.
`operational_state`	X		Possible values are EMAgent::disabled and EMAgent::enabled
`administrative_state`	M	X	Possible values are EMAgent::locked, EMAgent::unlocked, or EMAgent::shuttingdown
`system_title`	M		OID of system title, e.g. “1.2.3.4”
`get_community_string`	M	X	e.g. “public” or “private”
`set_community_string`	M	X	e.g. “public” or “private”
`transport_address`	M	X	IP address of the system associated with the Internet agent, specified as a string, such as “34.254.129.23”. An optional port number may be appended, such as “34.254.129.23:5723”
`management_protocol`	M		Internet management protocol used by the proxy to manage devices. Possible values are `EMSnmpAgent::snmp_v1` and `EMSnmpAgent::snmp_v2`.
`supported_mibs`	M	X	The names of the MIBs that the SNMP agent supports.
`access_control_enforcement`	M	X	Indicates where access control is applied: at the Internet agent, the ISO / Internet proxy, or both. Possible values are EMSnmpAgent::agent, EMSnmpAgent::proxy, or EMSnmpAgent::both.
`access_control_mechanism`	X	X	Indicates whether no access control, Internet access control as specified in [SNMPv2SEC], or ISO / CCITT access control as specified in [ISO10164-9] is to be used. Possible values are EMSnmpAgent::no_access_control, EMSnmpAgent::internet, or EMSnmpAgent::iso.

Example

```
RWBoolean
create_snmp_agent(
    const RWCString& system_name,
    const RWCString& snmp_agent_name
)
{
    EMSnmpAgent snmp_agent(
           EMSnmpAgentDn(system_name,snmp_agent_name));
    snmp_agent.set_administrative_state(EMAgent::unlocked);
    snmp_agent.add_supported_mib(“IIMCRFC1213-MIB”);
    snmp_agent.add_supported_mib(“IIMCSUN-MIB”);
    snmp_agent.set_get_community_string(“public”);
    snmp_agent.set_set_community_string(“private”);
    snmp_agent.set_transport_address(“123.234.34.23:2354”);
    snmp_agent.set_management_protocol(EMSnmpAgent::snmp_v1);
    snmp_agent.set_access_control_enforcement(
           EMSnmpAgent::agent);
    snmp_agent.set_access_control_mechanism(
           EMSnmpAgent::internet);
    snmp_agent.set_system_title(“1.2.3.4”);
    EMStatus status;
    if (!(status = snmp_agent.create_with_all_attributes())) {
           cerr << “Error: “ << status << endl;
           return FALSE;
    }
    return TRUE;
}
```

Enumerations

```
    enum EMSnmpAgent::Attribute {
         dn = EMAgent::num_attributes,
         system_title,
         get_community_string,
         set_community_string,
         transport_address,
         supported_mibs,
         management_protocol,
         access_control_enforcement,
         access_control_mechanism,
         num_attributes
    };
```

These are the attributes specific to EMSnmpAgent, in addition to the attribute defined in EMAgent which are common to EMCmipAgent, EMRpcAgent, and EMSnmpAgent.

```
    enum EMSnmpAgent::AccessControlEnforcement {
         agent=1,
         proxy=2,
        both=3,
         min_access_control_enforcement = agent,
         max_access_control_enforcement = both,
         num_access_control_enforcements
    };

    ostream& operator<<(
        ostream& s,
        const EMSnmpAgent::AccessControlEnforcement& enforcement
    );
```

```
enum EMSnmpAgent::AccessControlMechanism {
     no_access_control=0,
     internet=1,
     iso=2,
     min_access_control_mechanism = no_access_control,
     max_access_control_mechanism = iso,
     num_access_control_mechanisms
};

ostream& operator<<(
    ostream& s,
    const EMSnmpAgent::AccessControlMechanism& mechanism
);
```

```
enum EMSnmpAgent::ManagementProtocol {
     snmp_v1,
     snmp_v2,
     num_management_protocols
};

ostream& operator<<(
    ostream& s,
    const EMSnmpAgent::ManagementProtocol& protocol
);
```

Constructors, Destructor, Assignment Operator

```
EMSnmpAgent(
    const EMSnmpAgentDn& snmp_agent_id
);

EMSnmpAgent(
    const EMSnmpAgent& snmp_agent
);

~EMSnmpAgent();
EMSnmpAgent& operator =(
    const EMSnmpAgent& other_agent
);
```

Access Methods

```
EMStatus get_dn(
    EMSnmpAgentDn& dn
) const;

EMStatus set_dn(
    const EMSnmpAgentDn& dn
);
```

```
EMStatus get_system_title(
    RWCString& system_title
) const;

EMStatus set_system_title(
    const RWCString& system_title
);
```

```
    EMStatus get_get_community_string(
        RWCString& get_communitry_string
    ) const;

    EMStatus set_get_community_string(
        const RWCString& get_community_string
    );

    EMStatus get_set_community_string(
        RWCString& set_community_string
    ) const;

    EMStatus set_set_community_string(
        const RWCString& set_community_string
    );
```

```
    EMStatus get_transport_address(
        RWCString& transport_address
    ) const;

    EMStatus set_transport_address(
        const RWCString& transport_address
    );
```

```
EMStatus get_supported_mibs(
    RWTValSlist<RWCString>& supported_mibs
) const;

EMStatus set_supported_mibs(
    const RWTValSlist<RWCString>& supported_mibs
);

EMStatus add_supported_mib(
    const RWCString& supported_mib
);

EMStatus remove_supported_mib(
    const RWCString& supported_mib
);
```

```
EMStatus get_management_protocol(
    EMSnmpAgent::ManagementProtocol& management_protocol
) const;

EMStatus set_management_protocol(
   EMSnmpAgent::ManagementProtocol management_protocol
);
```

```
EMStatus get_access_control_enforcement(
   EMSnmpAgent::AccessControlEnforcement&
       access_control_enforcement
) const;

EMStatus set_access_control_enforcement(
   EMSnmpAgent::AccessControlEnforcement
       access_control_enforcement
);
```

```
    EMStatus get_access_control_mechanism(
        EMSnmpAgent::AccessControlMechanism&
            access_control_mechanism
    ) const;

    EMStatus set_access_control_mechanism(
        EMSnmpAgent::AccessControlMechanism
            access_control_mechanism
    );
```

Related Global Operators

```
    ostream& operator<<(
        ostream& s,
        const EMSnmpAgent& agent
    );
```

The stream output operator << is defined to provide an easy was to print out the value of `EMSnmpAgent`

Index