EMC® Smarts®
Perl Reference Guide
7.0

P/N 300-003-969
REV A01

EMC Corporation
Corporate Headquarters:
Hopkinton, MA 01748-9103
1-508-435-1000
www.EMC.com

Copyright © 2003, 2006 EMC Corporation. All rights reserved.

EMC believes the information in this publication is accurate as of its publication date. The information is subject to change without notice.

THE INFORMATION IN THIS PUBLICATION IS PROVIDED "AS IS." EMC CORPORATION MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND WITH RESPECT TO THE INFORMATION IN THIS PUBLICATION, AND SPECIFICALLY DISCLAIMS IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Use, copying, and distribution of any EMC software described in this publication requires an applicable software license.

For the most up-to-date listing of EMC product names, see EMC Corporation Trademarks on EMC.com

All other trademarks used herein are the property of their respective owners.

Contents

Preface v

Chapter 1 Introduction

Fundamental concepts 1-2
Overview of a simple API Perl script 1-4
Using primitives and object references 1-6
Event subscription 1-7

Chapter 2 Object

Description 2-2
Functions and methods 2-4

Chapter 3 Session

Description 3-2
Function groups 3-3
Error handling 3-4
Session management functions 3-5
Utility functions 3-11
Compatibility functions 3-12

Chapter 4 Primitives

Primitive naming conventions 4-2
Primitive calling conventions 4-3
Error handling 4-4
Error codes 4-5
Data types 4-6
Primitives 4-10

Index I-1

Preface

The EMC Smarts Perl API provides a programmable interface with EMC Smarts Domain Managers. You can use it to perform various actions such as:

- Getting information about a running Domain Manager
- Creating or deleting objects in the domain topology
- Getting or setting data about objects in the topology
- Executing operations that are declared for an object class

This manual, the *EMC Smarts Perl Reference Guide*, describes how to use the EMC Smarts Perl API to create Perl scripts that interact with running Domain Managers. It provides a reference to the API modules and includes sample scripts that put these modules to work.

Audience for this guide

This document is intended for software developers using the Perl programming language to implement EMC Smarts subscription and adapter applications.

Prerequisites

It is beneficial to have an understanding of the basic EMC Smarts platform operation, functionality, and architecture, and the MODEL programming language, which are available through the *EMC Smarts Educational Services* tutorials.

Proficiency in the Perl programming language is required, along with familiarity with the language's development environment and tools.

Readers of this document should also be familiar with basic programming and communication concepts and software development methodology, especially as they relate to distributed systems.

EMC Smarts installation directory

In this document, the term BASEDIR represents the location where EMC Smarts software is installed.

- For UNIX, this location is: /opt/InCharge<*n*>/<*productsuite*>
- For Windows, this location is: C:\InCharge<*n*>\<*productsuite*>.

The <n> represents the software platform version number. The <productsuite> represents the EMC Smarts product suite to which the product belongs. This location is referred to as BASEDIR/smarts.

Optionally, you can specify the root of BASEDIR to be something other than:

- UNIX: /opt/InCharge7
- Windows: C:\InCharge7

However, you cannot change the <productsuite> location under the root directory.

For more information about the directory structure of EMC Smarts software, refer to the *EMC Smarts System Administration Guide*.

Conventions used in this guide

EMC uses the following conventions for notes and cautions.

Note: A note presents information that is important, but not hazard-related.

IMPORTANT

An important notice contains information essential to the operation of the software.

CAUTION

A caution contains information essential to avoid data loss or damage to the system or equipment. The caution may apply to hardware or software.

Typographical conventions

EMC uses the following conventions in this guide:

Normal font	In running text: • Interface elements (for example, button names, dialog box names) outside of procedures • Items that the user selects outside of procedures • Java classes and interface names • Names of resources, attributes, pools, Boolean expressions, buttons, DQL statements, keywords, clauses, environment variables, filenames, functions, menu names, utilities • Pathnames, URLs, filenames, directory names, computer names, links, groups, service keys, file systems, environment variables (for example, command line and text), notifications
Bold	In procedures: • Names of dialog boxes, buttons, icons, menus, fields, and maps • Selections from the user interface, including menu items, and field entries • Key names • Window names • Code examples • Commands • User interface tabs
Italic	Used for: • Full publications titles referenced in text In procedures: • Directory paths • Filenames • Scripts • Important/unique words • URLs • Utilities • Variables

`Courier`	Used for: • System output • User entry
< >	Angle brackets used to enclose parameter or variable values supplied by the user.
[ ]	Square brackets used to indicate optional values.
|	Vertical bar used to indicate alternate selections. The bar means “or”.
{ }	Braces used to indicate content that you must specify (that is, x or y or z).
...	Ellipses used to indicate nonessential information omitted from the example.
%	Used to indicate a C shell prompt.
#	Used to indicate a C shell superuser prompt.
▼▲	Used to indicate a command is wrapped over one or more lines. The command must be typed as one line.

Pathname conventions

Directory pathnames are shown with forward slashes (/). Users of the Windows operating systems should substitute back slashes (\) for forward slashes.

Graphical conventions

If there are figures illustrating consoles in this document, they represent the consoles as they appear in Windows. Under UNIX, the consoles appear with slight differences. For example, in views that display items in a tree hierarchy such as the Topology Browser, a plus sign displays for Windows and an open circle displays for UNIX.

Smarts Manager

Unless otherwise specified, the term Smarts Manager is used to refer to EMC Smarts programs such as Domain Managers, Global Managers, and adapters.

Related documentation

In addition to this document, EMC Corporation provides a Help system for command line programs as well as product documentation.

Help for command line programs

Descriptions of command line programs are available as HTML pages. The index.html file, which provides an index to the various commands, is located in the BASEDIR/smarts/doc/html/usage directory.

EMC Smarts documentation

Readers of this guide may find the following related documentation helpful. It can be found in the BASEDIR/smarts/doc/pdf directory.

Note: These documents are updated periodically. Electronic versions of the updated manuals are available on the Powerlink website: **http://Powerlink.EMC.com**.

- *EMC Smarts Documentation Catalog*
- *EMC Smarts System Administration Guide*
- *EMC Smarts ICIM Reference*
- *EMC Smarts Common Information Model Infrastructure Models Chart*
- *EMC Smarts Common Information Model Application/Business Models Chart*
- *EMC Smarts ASL Reference Guide*
- *EMC Smarts Perl Reference Guide*

Software Development Kit Documentation

The following EMC Smarts documents are relevant to users of the Software Development Kit:

- *EMC Smarts Software Development Kit Remote API for Java* (in HTML format)
- *EMC Smarts Remote API Programmer's Guide*
- *EMC Smarts MODEL Reference Guide*
- *EMC Smarts MODEL Reference Guide* (in HTML format)

Refer to the *EMC Smarts Documentation Catalog* for documentation resources provided with other EMC Smarts product suites.

Suggestions for searching PDF files

You may search across multiple PDF files using the Adobe Acrobat Reader software.

1. If the documentation is not accessible to all users of the EMC Smarts product suite, copy the contents of the *BASEDIR/smarts/doc/pdf d*irectory to a central location, such as a shared drive on your LAN, that operators and others may view.
2. To search throughout the documentation library, open the Acrobat Reader software.
   1. Choose **Edit** > **Search**, and enter a word or phrase.
   2. In the **Where would you like to search** option, select **All PDF Documents in** and type the pathname of the location where the PDF documents reside.

If you have more than one EMC Smarts product suite installed, you can set up cross-product document searches by copying files from the BASEDIR/smarts/doc/pdf directory for each product suite into this common documentation directory path.

Where to get help

EMC support, product, and licensing information can be obtained as follows:

Product information — For documentation, release notes, software updates, or for information about EMC products, licensing, and service, go to the EMC Powerlink™ website (registration required) at:

`http://Powerlink.EMC.com`

Technical support — For technical support, go to EMC Customer Service on Powerlink. To open a service request through Powerlink, you must have a valid support agreement. Please contact your EMC sales representative for details about obtaining a valid support agreement or to answer any questions about your account.

Sales and customer service contacts — For the list of EMC sales locations, please access the EMC home page at:

`http://EMC.com/contact`

1

Introduction

The Remote Application Programming Interface (API) for Perl allows developers to create Perl scripts that connect to EMC Smarts servers as clients to exchange information, manipulate data, or drive EMC Smarts Domain Manager actions.

The API provides access to all EMC Smarts Domain Manager features using a syntax and logic that mirrors what is available through the Adapter Scripting Language (ASL) and the dmctl utility in a way that is natural to Perl developers.

Note: This API is compatible with EMC Smarts versions 6.0 and later.

The API runs on Windows 2000 and UNIX platforms that support Perl 5.6.1 and Perl 5.8.0 and the IO::Socket module.

This chapter contains the following information:

- Fundamental concepts 1-2
- Overview of a simple API Perl script 1-4
- Using primitives and object references 1-6
- Event subscription 1-7

Fundamental concepts

In order to create scripts that interact with an EMC® Smarts® Domain Manager, it is necessary to understand how the manager is configured.

The script creates, deletes, and interacts with instances of objects defined using the MODEL language. The MODEL language is an object-oriented language used to construct a data model to describe a managed domain. The language is used to define a set of classes and the attributes, relationships, operations, and events that are associated with the classes.

Classes

Classes describe the objects that are modeled for use in an EMC Smarts Domain Manager. For example, Router is a class, and all routers that are managed by an EMC Smarts domain are represented as Router objects in the domain. Every object in a class shares the same set of attributes, although the values of the attributes differ. Hence, every router has an IP address, an attribute, but the actual addresses are different. PowerSupply is also a class, but power supplies do not have IP addresses. However, the event power outage is relevant to the PowerSupply class but not to routers. Therefore, a model Class is a grouping of all objects that are similar in nature but not in detail.

Every model class has a number of properties, events and operations defined for them. The API provides functions for obtaining details of these definitions.

Instances

Object instances are specific occurrences of a class. For example, a class might describe a human and an instance of the class could be an object named Bill.

Properties - Attributes and relationships

Every instance in an EMC Smarts domain has a set of properties associated with it. These are values that describe the object. There are two distinct types of class properties supported by EMC Smarts.

1. Attributes

Attributes describe a class and for an instance of the class include information about its present state. Examples of attributes include an element's name and a counter that counts the number of packets traversing an interface. Attributes are simple strings, integers, booleans, or enumerations.

2. Relationships

Relationships define how instances are related to other instances. Relationships can be one-to-one, one-to-many, many-to-one or many-to-many. When only a single instance can be related to another instance, or instances, it is known as a relationship. When multiple instances can be related to another instance, or instances, it is known as a relationshipset.

Operations

Operations are actions that are specific to a class of object. For example, it makes sense to get the associated network adapter name for a MAC address but not for a router.

The API provides a mechanism for invoking these class-specific actions, passing information to them via arguments, and obtaining the results of the action.

Events

Events describe the failures that can occur for a class, the symptoms these failures cause, and the effect of failures. Symptoms can be local, observed in the instance of the class, or propagated, observed in instances related to the failing instances.

Primitives - Basic Domain Manager interface

Primitives are Perl functions that provide the basic interface between a client application and the EMC Smarts Domain Manager.

A number of these are likely to be used directly by scripts and will be familiar to ASL developers. These include getInstances(), getChildren(), getExplainedBy().

Others are normally hidden from view because higher level features can be used instead, which ultimately call the primitives. For example, primitives that are not normally used directly are *get()* and *invoke()*. These are the calls that allow an instance's properties to be queried and its operations to be called. In both the API for Perl and ASL, these calls are normally invoked by using a classic object-oriented syntax.

For more information about the EMC Smarts data structures, refer to the *EMC Smarts ASL Reference Guide*.

The Perl API is implemented as a set of Perl modules, which individual Perl scripts may access via the familiar 'use' directive. InCharge::session and InCharge::object offer the principal interface, respectively providing connection sessions to Domain Managers and access to objects within those managers. Simultaneous sessions to multiple Domain Managers may be established within a single Perl script. Properties and methods of objects within those managers may be accessed as with C++, offering somewhat broader functionality than that afforded by ASL.

Overview of a simple API Perl script

The general approach to writing a script using the Remote Perl API is to follow these basic steps:

1. Open a session.

   Initialize an EMC Smarts session, and obtain a reference to it (using either InCharge::session->init() or InCharge::session->new(), as appropriate).

   ```
   use InCharge::session;
   $session = InCharge::session->init( );
   ```

   You would choose the init() method if you want the script's user to supply the session connection details on the command line (such as the broker, domain name, username and password). The new() method can be used if you want your script to have greater control over the details used to establish the connection.

2. Work with the domain.

   Call the primitives required using the session reference obtained in step 1, and manipulate the data. For example,

   ```
   foreach $class ( sort $session->getClasses() ) {
       foreach $inst (
           sort $session->getInstances($class))
       {
           print $class . "::" . $inst . "\n”;
       }
   }
   ```

3. Close the session.

   Once the script has finished working with the domain, the session should be closed.

   ```
   $session->detach( );
   ```

Where access to the operations or properties of domain objects (such as Routers and Interfaces) is required, you use the features of the InCharge::object module. The script obtains an InCharge::object reference, and then uses it to access the required information. For example.

1. Establish an EMC Smarts session.

   ```
   use InCharge::session;
   $session = InCharge::session->init();
   ```

2. Obtain an object reference.

   Before an object in the domain can be accessed, the script needs to obtain an InCharge::object reference to the object of interest using the object() method of the session handle.

   ```
   $obj = $session->object( "Router::gw1" );
   ```

3. Manipulate the object.

   The reference obtained in step 2 can now be used to access the properties and operations of the object. Properties can be accessed using Perl's hashing syntax and operations can be invoked using Perl's object-oriented syntax conventions.

   ```
   $type = $obj->{Type};
   $obj->{Vendor} = "Cisco";
   $fan1 = $obj->findFan( 1 );
   ```

4. Close the EMC Smarts session.

   As before, the session should be closed when no longer required.

```
$session->detach( );
```

Using primitives and object references

The API provides function calls for accessing all the low-level facilities of EMC Smarts Domain Managers. Each of these primitives can be invoked with reference to the *InCharge::session* handle (see above), and takes arguments that exactly match the API syntax.

The API also provides an object-oriented abstraction layer that allows Perl code to access the Domain Manager using a syntax that is very similar to ASL. For example, in ASL you can list the vendors of all routers using this logic.

```
routers = getInstances( "Router" );
foreach router ( routers ) {
    obj = object( "Router", router );
    vendor = obj->Vendor;
    print( router . " - " . vendor );
}
```

When using the Perl API to perform the same action, the code looks like this.

```
@routers = $session->getInstances( "Router" );
foreach $router ( @routers ) {
    $obj = $session->object( "Router", $router );
    $vendor = $obj->{Vendor};
    print $router . " - " . $vendor . "\n”;
}
```

The two code fragments above are very similar. The first main difference is a matter of syntax. Perl uses ``$'' and ``@'' to denote scalar and array variables, and {..} to denote object properties, which are hash table lookups.The second difference is that the object(..) and getInstances(..) functions are called with reference to a session handle in the Perl code.

Event subscription

The Perl API provides mechanisms for subscribing to and acting upon events generated by EMC Smarts servers.

The EMC Smarts programming model delivers two different modes of client/server communication. The most direct is where a client makes a request of the EMC Smarts Domain Manager which acts on the request and responds. A simple example of this is an object query or update. For example, the action of obtaining the vendor of a particular device is one such query. In Perl, this query would be encoded in a manner similar to the following fragment.

```
use InCharge::session; use InCharge::object;
$session = InCharge::session->init( );
$device = "Router::gw1";
$obj = $session->object( $device );
$vendor = $obj->{Vendor};
print $vendor . "\n”;
```

The second mechanism provides asynchronous notifications via subscriptions and is used when the client program needs to listen for events generated by the EMC Smarts Domain Manager in response to other external events. One example would be a script that waits for the Vendor field of a particular router to change. In Perl, this could be coded in the following way.

```
use InCharge::session; use InCharge::object;
$session = InCharge::session->init( );
$observer = $session->observer( );
$device = "Router::gw1";
$session->propertySubscribe($device, "Vendor", 30 );
while ( 1 ) {
   @event = $observer->receiveEvent( );
   print "Vendor $event[2]::$event[3] is now \
                    $event[5]\n”;
}
```

The following sections provide an overview of mechanisms for creating and controlling a number of different types of subscriptions.

Registering an observer

In order to allow EMC Smarts servers to send subscribed events to a client program, the client must first register itself with the EMC Smarts Domain Manager as an event observer.

In Perl, this is done using the observer() method of the InCharge::session module. There are two steps needed to perform this action.

1. Connect to the Domain Manager and get a valid session object handle.

   First, the client must connect to the Domain Manager, establishing a new InCharge::session connection. This is done using either the InCharge::session->new or InCharge::session->init methods, depending on the level of control that the script requires to impose on the connection process. The result of this step, whichever procedure call is used, is a valid InCharge::session object handle.

   Here are some example code fragments that achieve this goal.

   ```
   $session = InCharge::Session->init( )
   ```

This parses the script command line for the --server, --broker and other standard options.

```
$session = InCharge::session->new(
    broker=>"192.168.0.3",
    domain=>"INCHARGE"
);
```

This allows the connection options to be specified explicitly by the script itself.

2. Attach an observer to the Domain Manager session.

   Once the script has obtained a handle that references the script/server connection ($session in the examples above), it can be used to obtain a second connection to the notification engine of the Domain Manager. This is obtained using the observer() method on the InCharge::session handle just obtained. The following code performs this action.

```
$observer = $session->observer();
```

   This $observer handle now references a second link between the client script and the Domain Manager, which is used to pass subscribed events to the client in real-time.

   The observer connection can be closed, detached, or destroyed using the detach() object method:

```
$observer->detach();
```

   This action has the side effect of cancelling any outstanding subscriptions.

Subscribing to notifications

Once the client has registered itself as an observer, the next step is to inform the Domain Manager about which events it is interested in receiving. EMC Smarts allows clients to subscribe to a number of different types of events. These are listed in Table 1-1.

Table 1-1 Subscription methods summary

Method type	Description	Method API call
property	Notifications about changes to specified object properties in the ICIM database. For example, when the “Vendor” field of Router::gw1 changes	propertySubscribe propertyUnsubscribe
topology	Notifications about changes to the topology, such as the creation and deletion of objects. This does not refer to object property changes.	topologySubscribe topologyUnsubscribe
event	Notifications about the posting and clearing of events and changes to their state.	subscribe unsubscribe subscribeAll unsubscribeAll getSubscriptionState IsSubscribed

This table gives the names of the methods used to subscribe to and unsubscribe from different types of notifications.

The following code segment is an example script that subscribes to changes of the Vendor field of every device in the topology.

```
$session = InCharge::session->init( );
$obs = $session->observer();
foreach $name ( $session->getClassInstances(
          "ICIM_UnitaryComputerSystem" ) ) {
   $session->propertySubscribe("::$name", "Vendor", 30);
}
```

Receiving notifications

Once the script has registered as an observer, and subscribed to the notifications of interest it then proceeds to listen for events and process them as required. The event reception method call is receiveEvent(). This returns an array of up to five values.

For the purposes of the descriptions that follow, we assume that events are returned in the array @event, as shown in the following script fragment.

```
@event = $observer->receiveEvent( );
```

Should the script require the event to be a single string with a separator used to delimit the fields, in the style of the ASL language, then the application can use the standard Perl join function:

```
$fs = "|";
$event = join( $fs, $observer->receiveEvent( ) );
```

The *receiveEvent()* method can take an optional parameter to specify a time-out in seconds, which may be fractional. If no event arrives within the specified time, a pseudo-event of type TIMEOUT is returned. For example,

```
@event = $observer->receiveEvent( 0.25 );
```

If no timeout is specified, the call waits forever.

The first element of the @event array, accessed using the Perl syntax: $event[0], contains the event's timestamp measured using normal UNIX time_t semantics; i.e., the number of seconds since midnight January 1, 1970.

The second element of the @event array, $event[1], contains a text string that describes the type of event received.

The array elements from $event[2] to $event[$#event] have meanings that depend on the semantics of the event type given in $event[1].

Event notification records

Event notifications are received from the Domain Manager when the status of an event changes. The format of the notification record is:

Table 1-2 Notification record - NOTIFY

Event record entry	Description
$event[0]	Timestamp (INTEGER)
$event[1]	“NOTIFY”
$event[2]	Object class name (STRING)
$event[3]	Object instance name (STRING)
$event[4]	Event name (STRING)
$event[5]	Event certainty (FLOAT)

Normally, the Domain Manager sends a single notification message when an event becomes active and a single clear message when the event is no longer active. If an event corresponds to a root-cause problem, it is possible that the certainty of the diagnosis will change over time. If the diagnosis certainty changes, the Domain Manager generates another notification. Notifications of this type are streamed in a slightly different manner. This difference in behavior is a feature of the front end Perl API, not the Domain Manager. The Domain Manager sends NOTIFY messages in both cases. The API keeps internal notes about active events, and changes the event type accordingly.

Table 1-3 Notification record - CERTAINTY_CHANGE

Event record entry	Description
$event[0]	Timestamp (INTEGER)
$event[1]	"CERTAINTY_CHANGE"
$event[2]	Object class name (STRING)
$event[3]	Object instance name (STRING)
$event[4]	Event name (STRING)
$event[5]	Event certainty (FLOAT)

When an event is cleared by the Domain Manager, the format of the record is:

Table 1-4 Notification record - CLEAR

Event record entry	Description
$event[0]	Timestamp (INTEGER)
$event[1]	"CLEAR"
$event[2]	Object class name (STRING)
$event[3]	Object instance name (STRING)
$event[4]	Event name (STRING)

Object create/delete records

An object create message is sent by the Domain Manager when a new object is created in the manager's repository. The format of an object create record is:

Table 1-5 Notification record - CREATE

Event record entry	Description
$event[0]	Timestamp (INTEGER)
$event[1]	"CREATE"
$event[2]	Object class name (STRING)
$event[3]	Object instance name (STRING)

An object delete message is sent by the Domain Manager when an object is deleted from the manager's repository. The format of an object delete record is:

Table 1-6 Notification record - DELETE

Event record entry	Description
$event[0]	Timestamp (INTEGER)
$event[1]	"DELETE"
$event[2]	Object class name (STRING)
$event[3]	Object instance name (STRING)

Class load records

A class load message is sent by the Domain Manager when a new class is created in the manager's repository. Classes are created when new MODEL-generated libraries are loaded. The format of a class load record is:

Table 1-7 Notification record - CLASS_LOAD

Event record entry	Description
$event[0]	Timestamp (INTEGER)
$event[1]	"CLASS_LOAD"
$event[2]	Class name (STRING)

Relation/property change records

A relation change message is sent by the Domain Manager when a relationship between objects changes. The format of a relation change record is:

Table 1-8 Notification record - RELATION_CHANGE

Event record entry	Description
$event[0]	Timestamp (INTEGER)
$event[1]	"RELATION_CHANGE"
$event[2]	Object class name (STRING)
$event[3]	Object instance name (STRING)
$event[4]	Relation name (STRING)

A property change message is sent by the Domain Manager when an object s property changes. The format of a property change record is:

Table 1-9 Notification record - ATTR_CHANGE

Event record entry	Description
$event[0]	Timestamp (INTEGER)
$event[1]	"ATTR_CHANGE"

Table 1-9 **Notification record - ATTR_CHANGE (continued)**

Event record entry	Description
$event[2]	Object class name (STRING)
$event[3]	Object instance name (STRING)
$event[4]	Property name (STRING)

Domain Manager connect/disconnect records

A Domain Manager disconnect record is generated when the connection to the server is lost. This differs somewhat from ASL operation. See the observer() primitive for proper handling of DISCONNECT events if 'restartableServer' operation is desired. These records are generated even if no subscriptions to the Domain Manager are issued. The format of the Domain Manager disconnect message is:

Table 1-10 **Notification record - DISCONNECT**

Event record entry	Description
$event[0]	Timestamp (INTEGER)
$event[1]	"DISCONNECT"
$event[2]	Domain name (STRING)

Note: There is no CONNECT record. In ASL, these are an artifact of the resartableServer front end that the Perl API does not provide. The restartableServer affords a means of invisibly attempting a (re)connection; the CONNECT message is an indication of success. The Perl API instead gives an immediate error on failure of *InCharge::session-->init()* or similar. It remains for the developer to provide retry logic to successfully CONNECT.

Subscription status records

When the Domain Manager receives a subscription request, it normally sends a notification back to the client to indicate whether or not the request was accepted. In the event of an error such as an invalid event name being specified, it does not report an error using normal Perl *die* semantics but uses a notification to report that the subscription was rejected. The format of the ACCEPT/REJECT message is:

Table 1-11 **Notification record - ACCEPT, REJECT, (PROPERTY)**

Event record entry	Description
$event[0]	Timestamp (INTEGER)
$event[1]	"ACCEPT" or "REJECT" or "PROPERTY_ACCEPT" or "PROPERTY_REJECT"
$event[2]	Object class name (STRING)
$event[3]	Object instance name (STRING)
$event[4]	Event or property name (STRING)

Event suspension records

Under certain circumstances, the Domain Manager will elect to suspend events if they are temporarily irrelevant. For example, when an aggregation contains no

triggering events a SUSPEND message is sent to the subscribed client. The format of the SUSPEND message is:

Table 1-12 Notification record - SUSPEND (PROPERTY)

Event record entry	Description
$event[0]	Timestamp (INTEGER)
$event[1]	“SUSPEND” or “PROPERTY_SUSPEND”
$event[2]	Object class name (STRING)
$event[3]	Object instance name (STRING)
$event[4]	Event or property name (STRING)
$event[5]	Descriptive message (STRING)

Time-out records

If no event arrives within the time specified as the optional argument to *receiveEvent()*, a TIMEOUT record is returned, whose message format is:

Table 1-13 Notification record - TIMEOUT

Event record entry	Description
$event[0]	Timestamp (INTEGER)
$event[1]	“TIMEOUT”
$event[2]	Domain name (STRING)

2

Object

The InCharge::object module provides the interface to EMC Smarts repository objects. With it you can:

- Get or set an object's attributes
- Invoke and object's operations

This chapter provides the following information:

- Description 2-2
- Functions and methods 2-4

Description

Name

InCharge::object - Access to EMC Smarts repository objects.

Synopsis

```
use InCharge::session;
$obj = $session->object( $class, $instance );
$value = $obj->get( "PropertyName" );
$value = $obj->{PropertyName};
$obj->put( "PropertyName", $value );
$obj->{PropertyName} = $value;
$rtn = $obj->invoke( "OperationName", .. arguments .. );
$rtn = $obj->OperationName( .. arguments .. );
```

The InCharge::object module allows objects in the EMC Smarts repository to be manipulated in an object-oriented style, similar to the ASL language.

Description

An InCharge::object reference is required to access the properties or methods of a Domain Manager object. This reference is returned from the object() or create() methods of the InCharge::session module. Both methods allow access to pre-existing objects but the create() method will also create the object if it does not already exist. While create() always accesses the Domain Manager, object() may or may not depending on the invocation technique.

```
$obj = $session->object( "Router", "edgert1" );
$obj = $session->object( "Router::edgert1" );
$obj = $session->create( "Router", "newrouter" );
```

Note: Whenever you specify the details of a repository instance to work within the API you have a choice of two syntaxes. You can either specify the object class and instance names as individual arguments or you can run them together with a delimiting double-colon (::). The API handles these two forms identically.

If you don't know the class to which an object belongs, you can either use a class argument of undef, or a string with nothing before the double-colon (::). For example,

```
$obj = $session->object( undef, "edgert1" );
$obj = $session->object( "::edgert1" );
```

The option of omitting the class name does not work with the InCharge::session->create() method because EMC Smarts cannot create an object without knowing which class to use. It does work with InCharge::session->object() and related calls because the process of referring to an existing instance can legitimately include a query to identify the object's class.

Note: If you choose not to provide the class name in these calls, the API does additional work to determine the object's class, imposing a slight performance penalty.

Once an object reference has been created, it can be used to invoke the object's operations or access its properties. Access to an object's attributes or properties can be obtained using calls shown in the following example.

```
$vendor = $obj->get( "Vendor" );
$vendor = $obj->{Vendor};
($vendor,$model) = $obj->get( "Vendor", "Model" );
%properties = $obj->get( );
```

```
$obj->put( "Vendor", "Cisco" );
$obj->{Vendor} = "Cisco";
$obj->put(Vendor => "Cisco", Model => "2010" );
```

These examples show that object properties can be accessed using either the get() and put() methods or via the psuedo-hash syntax. The latter syntax is preferred because it is closer to the original EMC Smarts built in ASL language logic.

Two special internal properties can be accessed using the hash syntax only. These give the name of the class and instance to which the object reference refers. Treat them as read-only fields.

```
$obj->{_class}           BUT NOT: $obj->get("_class")
$obj->{_instance}        BUT NOT: $obj->get("_instance")
```

Object operations can be invoked using the invoke() method, or directly, as in the example:

```
@ports = $obj->invoke( "findPorts" );
@ports = $obj->findPorts();
$port = $obj->
   invoke( "makePort", "1.0", "PORT-rt1/1.0", "Port" );
$port = $obj->makePort( "1.0", "PORT-rt1/1.0", "Port" );
```

Again, the latter syntax, calling the operation directly, is preferred.

Use the invoke() method to access an object operation that duplicates the name of any of the built-in methods of the InCharge::object class. The first of these two calls the new() operation of the object in the repository, whereas the second calls the built-in new*()* method of the *InCharge::object* class.

```
$obj->invoke( "new", "qtable" );
$obj->new( "qtable" );
```

Note that InCharge::object is used for accessing ICIM instance operations and properties only. If you wish to make other ICIM calls that refer to instances, such as subscribe(), use the features of InCharge::session directly. It is invalid to code:

```
$obj->propertySubscribe( "Vendor" );
```

Instead, use:

```
$session->propertySubscribe($class, $instance, "Vendor");
```

or,

```
$session->propertySubscribe( $obj, ``Vendor'' );
```

This is because the propertySubscribe is not a repository class operation but a primitive.

```
dmctl -s DOMAIN getOperations CLASSNAME | more
```

Likewise, to determine what properties can be accessed using this module use:

```
dmctl -s DOMAIN getProperties CLASSNAME | more
```

Functions and methods

object

```
$object = $session->object( $class, $instance );
```

Creates a new object reference.

get

```
$result = $obj->get( $property_name [, $property_name ...] )
```

or,

```
$result = $obj->{$property_name};
```

or,

```
%all_properties = $obj->get( )
```

Gets the value for the specified `property(s)` of the object.

The type of return value depends on the calling syntax used get() or hash() and the Perl evaluation context, scalar or array, as shown in the following table.

Table 2-1 Return type for call syntax and Perl evaluation context

Expression syntax	Property type	Return type in scalar context	Return type in array context
$obj->{prop}	scalar	scalar	scalar in [0]
$obj->{prop}	array	array ref	array ref in [0]
$obj->get("prop")	scalar	scalar	scalar in [0]
$obj->get("prop")	array	array ref	array

Multiple values are always returned in an array or array reference.

To access the content of a property whose name is held in a variable, use the Perl typical logic, as shown.

```
$propname = "Vendor";
$value = $obj->{$propname};
```

You can also get multiple values in a single get() call by listing all the property names as arguments. The results are returned in an array. With version 6 or later of EMC Smarts products, this is faster than using multiple single-property get() calls. On servers before version 6; there is no difference.

```
( $vendor, $type ) = $obj->get( "Vendor", "Type" );
```

You can also call get() with no arguments, in which case it creates a hash containing all the object properties and relations. There is no syntactical advantage, but there is a significant speed advantage on version 6 or later of EMC Smarts products.

See also the get_t() call which extends the functionality of this method by returning additional information, to identify the type of data held in the property.

Note: The get() and get_t() primitives throw an error when used to access a nonexistent property or one that cannot be retrieved for any reason, whereas the psuedo-hash syntax simply returns an undef value. This difference allows the Data::Dumper logic to display an entire object without erring even when some properties cannot be retrieved.

get_t

```
($type, $value) = $obj->get_t( $property_name );
```

or,

```
@types_and_values =
      $obj->get_t( $prop1 [, $prop2 [, prop3 .. ] ] )
```

or,

```
%all_property_types_and_values = $obj->get_t( );
```

This is like the get() method, except that it returns the type of the return value as well as the value itself. The data types are encoded as integer numbers. If the return is an array, then $value will receive a reference to the array. If the return is a scalar, then *$value* will hold it. $session->TYPE() can be used to convert the $type integer value to a mnemonic string.

The second syntax gets the types and values for multiple properties. Each type/value pair is held in a 2-element sub-array within the returned data.

The third syntax gets the types and values for all the properties and relations of the object and stores them in a hash, indexed by the property names. This approach has a significant performance benefit when working with version 6 or later of EMC Smarts products.

Example:

```
$obj = $session->object( "Router::gw1" );
( $type, $value ) = $obj->get_t( "Vendor" );
print "Vendor value ='$value', \
       type = ".$session>TYPE($type)."\n”;
```

This example will print:

```
Vendor value='CISCO', type = STRING
```

put

```
$object->put( $property_name, $value );
```

This method allows fields of the object to be modified in the EMC Smarts repository.

This method is used in a number of ways, however, the use of the pseudo-hash syntax is the preferred option for syntactic equivalence with the EMC Smarts native ASL language.

```
$obj->put( "Vendor", "Cisco" );
$obj->{Vendor} = "Cisco";
$obj->{ComposedOf} = [ ];
```

To set more than one property in a single call, use multiple name:value pairs, such as:

```
$obj->put(Vendor => "Cisco",
       PrimaryOwnerContact => "Joe Bloggs");
```

or

```
%updates = ( Vendor => "Cisco",
       PrimaryOwnerContact => "Joe Bloggs");
$obj->put( %updates );
```

When using either syntax to set a relationship or list property, use a reference to a Perl array, such as:

```
$obj->{ComposedOf} = [ $a, $b, $c ];
$obj->put( "ComposedOf", \@things );
```

Use *insertElement()* and *removeElement()* to add or remove elements from a list.

isNull

```
$boolean = $object->isNull();
```

Tests to see whether the object is present in the repository.

TRUE means that the object is not present. FALSE means it is present.

invoke

```
reply = $object->invoke( $operation, ... arguments ... );
```

Invokes the named repository class operation on the object.

The arguments passed should be as expected by the operation. If the operation returns a scalar value, the call should be called in a scalar context. If it returns an array, it should be invoked in an array context.

Note: The preferred way of achieving the same result is to use the operation name directly. Thus, the following are equivalent but the latter is preferred.

```
$obj->invoke( "makePort", "1.0", "First port", "Port" );
$obj->makePort( "1.0", "First port", "Port" );
```

Also, see the *invoke_t()* method, described below.

invoke_t

```
( $type, $value ) =
         $object->invoke_t($operation, .. args .. )>
```

Invokes the named class operation on the object in the same way as invoke(), but invoke_t() also returns the type of data returned by the call.

The data types are encoded as integer numbers. If the return is an array, then the $value will receive a reference to the array. If the return is a scalar, then $value will hold it.

insertElement

```
$obj->insertElement( $relation, @object[s] );
```

Inserts the specified objects into an object relationship.

One or more can be specified to be inserted.

```
$obj->insertElement( "ComposedOf",
          "Interface::IF-ether1",
          "Interface::Loopback/0");
$obj->insertElement( "ComposedOf", @interfaces );
```

removeElement

```
$obj->removeElement( $relation, @item[s] );
```

Removes the specified items from an object relationship.

One or more items can be specified to be removed.

```
$obj->removeElement( "ComposedOf",
          "Interface::IF-ether1",
          "Interface::Loopback/0" );
$obj->removeElement( "ComposedOf", @interfaces );
```

delete

```
$obj->delete( )
```

Deletes the specified item from the repository, but without performing any clean-up of inter-object dependencies.

Consider using the *remove()* operation, if one exists, instead for a more complete action.

notify

```
$obj->notify( $event_name );
```

Notifies the specified event for the object.

```
$objref->notify( "Unresponsive" );
```

clear

```
$obj->clear( $event_name );
```

Clears the specified event for the object.

```
$objref->clear( "Unresponsive" );
```

countElements

```
$count = $obj->countElements( $relation )
```

Counts the number of elements in the given relationship or throws an error if $relation is not a relationship.

```
$count = $obj->countElements( "ComposedOf" );
```

3

Session

The InCharge::session module provides the interface to EMC Smarts Domain Managers. With it you can:

- Attach to a Domain Manager
- Create and destroy objects
- Invoke operations
- Subscribe to Domain Manager events

This chapter provides the following information:

- Description 3-2
- Function groups 3-3
- Error handling 3-4
- Session management functions 3-5
- Utility functions 3-11
- Compatibility functions 3-12

Description

Name

InCharge::session - EMC Smarts Domain Manager session management.

Synopsis

```
 use InCharge::session;
 $session = InCharge::session->init( );
 $session = InCharge::session->new( "INCHARGE" );
 $session = InCharge::session->new(
         broker=>"localhost:426",
         domain=>"INCHARGE",
         username=>"noddy",
         password=>"bigears",
         traceServer => 1
 );
 $object = $session->object( "Host::toytown1" );
 $object = $session->create( "Router::crossroads" );
(... and continuing with the methods, described below ...)
```

Description

This module provides the mechanisms for accessing an EMC Smarts Domain Manager in a manner that is similar to that employed by the EMC Smarts ASL language. It provides the main access point to EMC Smarts domains, allowing scripts to establish client/server connections and to obtain InCharge::object references that can be used to manipulate the objects in the topology.

Refer to Chapter 1, “Introduction,” for an overview of this and the other InCharge::* modules and a simple tutorial description of how they are used.

Function groups

InCharge::session provides access to four kinds of functions.

Session management

Functions in this group are the principle functions of the module. They are used for managing the Perl client/Domain Manager connection. Using these functions a script can attach, detach, listen for events, and create InCharge::object references.

EMC Smarts primitives

The InCharge::session module permits access to the low-level primitive functions of the Domain Manager, allowing actions such as getClasses() and getInstances() to be performed. These primitives do not all exactly mirror the interface provided by dmctl or the native ASL language. For example, dmctl has a save command that does not have an exact primitive equivalent, but there are two primitives that can be invoked to give the same results. These are storeClassRepostity() and storeAllRepository(). Where primitives exist that semantically match dmctl or ASL commands but differ in name, aliased names are provided to give syntactic compatibility.

Utility functions

This group includes functions to provide additional logical assistance to writers of EMC Smarts Perl scripts.

Compatibility functions

This group of functions provides wrappers around the primitives to provide an interface that is more consistent with the EMC Smarts native ASL language and dmctl utility.

Wrapper functions of this type are provided only for functions where the syntax and semantics of the primitive are not compatible with ASL or dmctl. The save example has already been sited, above, to highlight one such function.

Error handling

Errors are reported back to the invoking script using Perl's *die* mechanism, and can be caught using eval. This is typical Perl coding practice and mimics the try-throw-catch logic of Java and C++.

Session management functions

The following session management functions are provided.

new

```
$session = InCharge::session->new( .. options .. );
```

This function establishes a connection between the calling Perl script and a Domain Manager and returns a tied reference that can be used thereafter to manipulate the domain and the entities contained in its repository.

If the domain name is the sole option passed, it can be specified without the domain=> key.

The username and password options (see below) are required if connecting to a SAM server or a Domain Manager with authentication features enabled. If neither of these arguments is given, the clientConnect.conf file is used to determine the username and password or the mechanism to obtain them.

Possible options are:

1. broker => $host[:$port]

   This specifies the EMC Smarts broker from which the domain details are to be lifted. The string consists of a hostname or IP address followed by an optional port number, delimited by a colon.

   The default host is localhost, and the default port is 426.

2. domain => [$host:$port/]$domain
3. server => [$host:$port/]$domain

   This specifies the name of the domain to be used. If the host and port details are also given, then the API does not refer to the broker to determine them.

   The default domain name is INCHARGE.

Note: The option name "server" can be used in place of "domain" and the two options have the same meaning.

4. user => $user_name
5. username => $user_name

   This specifies the name of the user to be used in connecting to the domain. If user or username is specified, then password must also be specified. If the username is not given, then the API refers to the clientConnect.conf file to determine the authentication information to use when establishing the connection.

   There is no default username.

   If no username is specified, the script inspects and interprets the SM_AUTHORITY environment variable in the same way that the main EMC Smarts software does and may prompt the user for the username and password using the standard I/O device.

6. password => $password

   This specifies the password for the user given with the username option.

Note: The username and password must both be supplied, or neither of them must be specified.

7. description => $description

   This describes the role of the script and is noted by the Domain Manager for use in debug and other logging messages. Its contents are not significant, otherwise. The default is Perl-Client.

8. traceServer => 1

   If specified and given a true value, non-zero, then server-level tracing is turned on. This causes the Domain Manager to log information about every primitive call invoked by the script, which can quickly fill up the server's log file. It is recommended to use this sparingly since it also has a negative impact on the Domain Manager's performance.

9. timeout => $timeout

   This specifies the timeout to be tolerated while waiting for responses from the Domain Manager to primitive requests. The default value is 120 seconds. Take care not to make this value too low. Otherwise, slow to-process requests will fail in a manner that looks like a communication link failure between the script and the Domain Manager.

init

```
$session = InCharge::session->init( );
```

This is the simplified version of InCharge::session->new(). It parses the script's command line, looking for options that specify the broker, Domain Manager username, password, and trace options. Then it invokes the primitive InCharge::session->new() with those arguments and passes back the result.

InCharge::session->init() looks for the following script command line arguments.

```
--broker=<brokerIP[:bokerPort]>   (also: -b)
--server=<domain-name>            (also: -s)
--user=<username>                 (also: -u)
--password=<password>             (also: -p)
--traceServer
--timeout
```

If neither the --user (or -u) and --password (or -p) are specified, the script makes use of the SM_HOME/conf/clientConnect.conf file to determine the username and password to be employed; see comments in the file for details of this mechanism. This mechanism is turned on by specifying the value <STD> for the SM_AUTHORITY environment variable.

If it encounters a command line syntax error, it calls usageError, in the main script, which the developer must provide. A single large text string containing a description of the standard options handled is passed as the argument to usageError, allowing the author to include information about the standard options as well as any nonstandard ones provided. If the usageError subroutine does not exist, a default error message is printed on STDERR.

Note that the init() function consumes (removes) the command line arguments it handles from @ARGV as it processes them; therefore you can access the @ARGV array after its execution to process additional arguments without needing to skip the

standard ones. However, you cannot use the init command twice in the same script without first saving and restoring the contents @ARGV, as in the example:

```
@SAVE = @ARGV;
$session1 = InCharge::session->init( );
@ARGV = @SAVE;
$session2 = InCharge::session->init( );
```

broken

```
$flag = $session->broken( );
```

Returns non-zero (TRUE) if the session with the Domain Manager is broken in some way.

This indicates connection or protocol failures. To continue working with a broken session, the script should call the reattach() function, and then re-establish the event subscription profiles required.

reattach

```
$session->reattach( );
```

Re-establishes a connection that has been detached or broken.

This can be called to reconnect to a server to which the connection has been lost. Re-establishing the connection does not automatically re-establish observer sessions, subscriptions, transactional or other session state information.

If the call is used to reattach a session that had an active observer, the observer connection is closed as a side effect of the action and must be reopened separately.

This function should be called after a [13] I/O Error is thrown by any of the Domain Manager access calls in order to shut down and reopen the socket, cleaning up the protocol. If this step is not taken there is a danger that residual packets on the connection would cause synchronization problems between the client and Domain Manager. "Error handling" on page 4-4 describes error prefixes, including the [13] prefix.

Note: The `reattach()` primitive doesn't return a new session identifier but refreshes the referenced one. This is not a *dup()* style of action.

detach

```
$session->detach( );
```

Detaches from the EMC Smarts domain referred to by $session.

This call can be used for either an EMC Smarts session, created using InCharge::session->new(), or an observer session, created using InCharge::session->observer().

If this is used to detach a session with an active observer, the observer is also closed.

This call does not completely destroy the $session reference contents but retains enough information to allow the session to be re-established. Thus, it is possible to call $session->reattach() to reconnect to the Domain Manager using the same

parameters as were used in the initial connection. However, the event subscriptions need to be re-established explicitly in this event.

observer

```
$observer_session = $session->observer( .. options .. );
```

Creates and returns a reference to a connection to a Domain Manager from which subscribed events can be receive.

This establishes a new socket between the client and Domain Manager. Once connected, events can be subscribed to using the various subscribe methods, and they can be received using:

```
@event_info = $observer_session->receiveEvent( );
```

Specifying the option connectEvents => 1 to the observer() function causes server disconnection to be notified as a DISCONNECT event rather than an [13] I/O Error. However, unlike ASL, the re-connection is not performed automatically. The script can use the $session->reattach() call to attempt an explicit re-connection and must then re-establish any event subscriptions and other contexts.

Specifying the option ignoreOld => 1 causes events generated before the connection was established to be discarded automatically. The use of this option is not generally recommended since the atomicity of time measurement on NT and UNIX makes its results somewhat unpredictable.

Repeated calls to the observer() method of a session return references to the same observer. It is not possible to create multiple observers on the same session.

receiveEvent

```
@event = $observer_session->receiveEvent( [ $timeout ] );
```

Listen for subscribed events from the Domain Manager.

The received events are returned as an array or, in scalar context, a reference to an array containing three or more elements. "Event subscription" on page 1-7 describes the different events to which you can subscribe.

The first element of all events is the timestamp, on the Domain Manager's system clock, and not the client's clock. The second element is a string defining the event type. The other elements are event-specific.

The $timeout is optional, and specifies a timeout period, in seconds, that the script is prepared to wait for an incoming event. If no event arrives in this time period, an event of type TIMEOUT is returned. The $timeout can be specified in fractions of a second, or "float"; for example, 0.25 = a quarter second.

object

```
$obj = $session->object( $objectname );
```

Creates a new InCharge::object reference that can be used to invoke methods of the InCharge::object module.

As an example, to obtain the value of the Vendor field for a particular object, use:

```
$obj = $session->object( "::gw1" );
$vendor = $obj->{Vendor};
```

You can even combine these into a single line, such as:

```
$vendor = $session->object( "::gw1" )->{Vendor};
```

The *$objectname* parameter can be specified in any of the following styles.

1. object('Router::gw1')

   A single string where both the class and instance names are specified with double colons (::) delimiting them. If variables are to be used to specify the relevant parts of the string, then it is important that at least the variable before double-colon (::) is encased in braces because without them, Perl will give the (::) characters its own meaning.

2. object('Router', 'gw1')

   An example of two strings with one for the class and one for the instance name.

3. object('::gw1')

   An example of one string with the class name missing. The API will make a query to the Domain Manager to discover the actual class for the object causing a minor performance penalty.

4. object(undef, 'gw1')

   An example of two parameters with the first one undefined. This also results in the API performing a Domain Manager query.

5. object('gw1')

   An example of a single parameter that doesn't include the double-colon (::) delimiter, which must contain just the instance name. As above, a Domain Manager query is performed to determine the relevant class name.

An important difference between the API and the native ASL language is that if you create an object, using *object()*, in native ASL without specifying the class name, the language assumes that the class MR_Object can be applied. This restricts the level of property and operation access that can be used. The API queries the repository to determine the actual class for the instance, giving complete access to the resulting object's features.

create

```
$obj = $session->create( $objectname );
```

Like object() above, the create() call creates an InCharge::object valid reference through which a specified instance can be manipulated. However, unlike object(), above, the create() method creates the object if it doesn't already exist.

Since it has the ability to create objects, it is important that the object name specified as an argument includes both the instance name and the class name. You cannot use the ::instance or (undef, $instance) syntaxes for specifying the object name. You can, however, use either the Class::Instance or ($class, $instance) syntax described for the object() method above.

Unlike the createInstance() primitive, it is not an error to call the create() method for an object instance that already exists. In this case, the call is equivalent to the $session->object() call above and it simply returns the InCharge::object valid reference to the instance.

callPrimitive

```
RESULT = $session->callPrimitive($primitiveName, @arguments)
```

Calls the specified Domain Manager primitive, passing it the arguments and returning its result.

Note: For most primitives, this is a complex invocation sequence. However, it is only actually needed when a primitive and a method of the *InCharge::session* module share the same name, and you wish to use the primitive version.

The following are equivalent, although the first is preferred.

```
@list = $session->getInstances( "Router" );
@list =
   $session->callPrimitive("getInstances", "Router");
```

The put() primitive is one of the few primitives where these two ways of calling it are not equivalent. This is because the InCharge::session module exports its own variant of the method. If you must gain access to the primitive version, you will need to use the callPrimitive() mechanism. However, this is not recommended, since the syntax is complex. See "put" on page 3-12 for more details.

The type of the RESULT in array or scalar context is dependant on the primitive being called. In general, if the primitive returns a scalar you get a scalar or, in array context, a single element array. If the primitive returns an array you get an array, in array context, or array reference, in scalar context.

Utility functions

TYPE

```
$number = $session->TYPE( $string );
$string = $session->TYPE( $number );
```

Converts a Domain Manager data type mnemonic string to its internal numeric code, or vise versa. So the following prints “13”.

```
print $session->TYPE( "STRING" ) . "\n";
```

The following code prints “STRING”.

```
print $session->TYPE( 13 ) . "\n";
```

getFileno

```
$fno = $session->getFileno( );
```

Returns the underlying system file number that refers to the socket used for the script/server connection. This is useful when the script wishes to use the Perl *select* statement to listen for events from multiple domain servers, using multiple observer objects.

getProtocolVersion

```
$ver = $session->getProtocolVersion( );
```

Returns the protocol version number supported by the Domain Manager. This is a single integer number derived by the following calculation.

```
( major * 10000 ) + (minor + 100) + revision
```

Hence, version “V5.1” is represented by the number 50100, and version “V4.2.1” is represented by 40201.

primitiveIsAvailable

```
$boolean = $session->primitiveIsAvailable( $primitive_name )
```

Checks whether the named primitive is available in the Domain Manager.

A value of 1 means that it is available, and value of 0 means that it is not available, either because it is an undefined primitive or was introduced in a later version of the Domain Manager software.

```
if ( $session->primtiveIsAvailable (
          "getMultipleProperties" ) {
    $vendor, $model ) = $session->getMultipleProperties (
                $obj, [ "Vendor", "Model" ] );
} else {
    $vendor = $obj->{Vendor};
    $model = $obj->{Model};
}
```

Compatibility functions

The following functions add varying degrees of wrapper logic round the EMC Smarts primitives, to make them more compatible with the native ASL language.

save

```
$session->save( $filename [, $class ] );
```

Saves the repository in the specified file. If a class name is specified then only the instances of that class are saved.

put

```
$session->put( $object, $property, $value );
```

It is not recommended that this method be used extensively. Instead, use the features of InCharge::object.

This method changes the value of an object property. This version differs from the *put_P()* primitive in that the latter requires the value type to be specified explicitly, whereas this version determines and caches the type. The following calls are, therefore, equivalent, although the first is preferred.

```
$obj = $session->object( "Router::gw1" );
$session->{Vendor} = "Cadbury";
$obj->put( "Vendor", "Cadbury" );
$obj->put( Vendor => "Cadbury" );
$session->put( "Router::gw1", "Vendor", "Cadbury" );
$session->object( "Router::gw1" )->{Vendor} = "Cadbury";
$session->callPrimitive( "put_P", "Router", "gw1",
          "Vendor",[ "STRING", "Cadbury" ] );
```

When giving a value to an array property, such as the ComposedOf relationship, pass an array reference as shown in the example:

```
$obj->{ComposedOf} = [
"Interface::IF-if1",
"Interface::IF-if2"
];
```

Also, you can set more than one property in a single call. This can reduce complexity in the script layout but has minimal performance advantage.

```
$obj->put(
    Vendor   => "CISCO",
    Model    => "2500",
    Location => "Behind the coffee machine"
);
```

invoke

```
RESULT = $session->invoke($object, $operation[, @arguments]);
```

It is not recommended that this method be used extensively. Instead, use the features of InCharge::object.

This method invokes the specified object operation, passing it the listed arguments and returning the RESULT.

The type of the RESULT in array or scalar context is dependant on the operation being called. In general, if it returns a scalar you get a scalar or, in array context, a single element array. If it returns an array you get an array, in array context, or array reference, in scalar context.

Note: This method's semantics and syntax differ from the primitive method *invokeOperation()* in that the latter needs to have the types of the arguments specified explicitly, whereas for this method the InCharge::session module version discovers and caches the operation argument types and does not require the arguments to be listed in arrays of array references.

Additional documentation about the operations that exist for a particular class can be obtained using the dmctl utility, as shown:

```
dmctl -s DOMAIN getOperations CLASSNAME
```

The following examples are equivalent; the first example is preferred.

```
$obj = $session->object( "Router::gw1" );
$fan= $obj->findFan( 2 );
$fan = $session->invoke( "Router::gw1", "findFan", 2 );
$fan = $session->callPrimitive( "invokeOperation",
"Router", "gw1", "findFan",
[ [ "INT", 2 ] ]
);
```

invoke_t

```
($type, $value) =
      $session->invoke_t( $object, $operation [, @arguments]
);
```

Identical to *invoke()* except that the return indicates both the type and the value of the returned data.

The value is a Perl scalar, if the operation returns a scalar, or an array reference, if the operation returns an array. The type will contain one of the Domain Manager internal type codes. For example, “13” is the code for a string.

findInstances

```
@instances =
      $session->findInstances( $c_patn, $i_patn [, $flags] )
```

or

```
@instances =
      $session->findInstances( "${c_patn}::${i_patn}" [, $flags] )
```

Finds instances that match the class and instance patterns, according to rules specified in the flags.

The $flags is a set of characters that modifies the way the call works.

A flag of “n” means that subclasses are not recursed into. Therefore, instances in matching classes only are returned. Without “n”, instances of matching classes and their subclasses are returned.

A flag of “r” means that UNIX-like RegEx matching is used during the search. If the “r” flag is not specified, the search uses glob pattern matching.

The default is no flags; i.e., glob matches and recursion.

Results are returned as a list of strings, each of which contains a class and instance name delimited with double-colon (::).

Note: The search strings are anchored as if the "^" and "$" had been used in the UNIX-style pattern. Therefore, "rr*" matches "rred" but not "herring", whereas "`*rr*" matches both of them.

Example:

```
@found = $session->findInstances( "Router::gw*", "n" );
```

getCauses

```
@events = $session->getCauses( $objectname, $event [, $oneHop] );
```

The getCauses() function returns a list of problems that cause an event.

The function arguments are class, instance (possibly combined into one), and event. The function returns the problems that cause the event based on the relationships among instances defined in the Domain Manager.

The oneHop parameter is optional. If it is omitted or passed as FALSE, the full list of problems explaining eventname, whether directly or indirectly, is returned. If it is passed as TRUE, only those problems that directly list eventname among the events they explain are returned.

The function returns an array of array references with the format:

```
[
    [ <classname::instancename>,<problemname> ],
    [ <classname::instancename>,<problemname> ],
    ...
]
```

Note: The class and instance names are returned as a single double-colon (::) delimited string, giving two strings per returned event in total. This is different from the native ASL language which returns the class and instance names separately, giving three strings for each event.

Example:

```
@causes =
    $session->getCauses( "Router::gw1",
           "MightBeUnavailable"
);
```

getClosure

```
@events = $session->getClosure($object, $eventname[, $oneHop]);
```

The getClosure() function returns a list of symptoms associated with a problem or aggregation.

The function returns the symptoms associated with the problem or aggregate based on the relationships among instances defined in the Domain Manager.

The oneHop parameter is optional. If it is omitted or passed as FALSE, the full list of problems explaining eventname, whether directly or indirectly, is returned. If it is passed as TRUE, only those problems that directly list eventname among the events they explain are returned.

The function returns an array of array references with the format:

```
[
    [ <classname::instancename>,<problemname> ],
    [ <classname::instancename>,<problemname> ],
    ...
]
```

Note: The class and instance names are returned as a single double-colon (::) delimited string, giving two strings per returned event in total. This is different from the native ASL language which returns the class and instance names separately, giving three strings for each event.

Example:

```
@symptoms =
    $session->getClosure( "Router::gw1", "Down", 0 );
```

getExplains

```
@events = $session->getExplains($object, $eventname[, $onehop ]);
```

MODEL developers can add information to a problem in order to emphasize events that occur because of a problem. The getExplains() function returns a list of these events.

The $onehop parameter is optional. If it is omitted or passed as FALSE, the full list of problems explaining $eventname, whether directly or indirectly, is returned. If it is passed as TRUE, only those problems that directly list eventname among the events they explain are returned.

The function returns an array of array references with the format:

```
[
    [ <classname::instancename>,<problemname> ],
    [ <classname::instancename>,<problemname> ],
    ...
]
```

Note: The class and instance names are returned as a single double-colon (::) delimited string, giving two strings per returned event in total. This is different from the native ASL language which returns the class and instance names separately, giving three strings for each event.

getExplainedBy

```
@events = $session->getExplainedBy($object, $event[, $onehop ]);
```

This function is the inverse of the getExplains() function.

It returns those problems which the MODEL developer has listed as explaining this problem.

The $onehop parameter is optional. If it is omitted or passed as FALSE, the full list of problems explaining $event, whether directly or indirectly, is returned. If it is passed as TRUE, only those problems that directly list $event among the events they explain are returned.

The function returns an array of array references with the format:

```
[
    [ <classname::instancename>,<problemname> ],
    [ <classname::instancename>,<problemname> ],
    ...
]
```

Note: The class and instance names are returned as a single double-colon (::) delimited string, giving two strings per returned event in total. This is different from the native ASL language which returns the class and instance names separately, giving three strings for each event.

subscribe and unsubscribe

```
$session->subscribe( $C, $I, $E [, $flags ] );
$session->subscribe( "$C::$I::$E[/$flags]" );
$session->unsubscribe( $C, $I, $E [, $flags ] );
$session->unsubscribe( "$C::$I::$E[/$flags]" );
```

These functions subscribe, or unsubscribe, to notifications of the specified events. “$C”, “$I”, “$E” must be regexp patterns representing the classes, instances, and events to which to subscribe.

The *unsubscribe()* function is the inverse of subscribe().

The $flags value is a bitwise combination of the values in the following table or a more mnemonic string, as shown in the table.

Table 3-1 Subscription flag parameter values

Flag bitfield value	Description
0x000001	Simple event
0x000002	Simple aggregation
0x000010	Problem
0x000020	Imported event
0x000040	Propagated aggregation
0x0000ff	All
0x001000	Expand subclasses
0x002000	Expand subclasses events
0x004000	Expand aggregations
0x008000	Expand closures
0x010000	Sticky
0x020000	Undo all
0x040000	Quiet accept
0x080000	Quiet suspend
0x100000	Glob

As a compatibility aid, the $flag can also be specified as a string of letters. In this case, each of the letters are subscription qualifiers: “p” means subscribe to problems, “a” means subscribe to aggregates (impacts), and “e” means subscribe to events. If none

of these are present, “p” is assumed. The letter “v” means run in verbose mode, which turns on subscription control messages. The action of these options is the same as that provided by the sm_adapter program’s --subscribe= option.

Examples:

```
$session->subscribe( "Router", ".*", ".*", "/pev" );
$session->subscribe( "Router::.*::.*/peav" );
$session->subscribe( $obj, ".*", 0x3 );
$session->unsubscribe( $obj, ".*", 0x3 );
```

transaction, abortTxn and commitTxn

```
$session->transaction( [ $flag ] );
$session->abortTxn( );
$session->commitTxn( );
```

These functions start, commit and abort transactions.

Using transactions, you can commit many changes to the objects in an Domain Manager as a single atomic “transaction” or choose to abort all of them. Use the following syntax to create a transaction:

```
$session->transaction();
```

After initiating the transaction, every change made to an object does not affect the object until you commit the transaction. If the transaction is aborted, any changes made will not affect the object. Use the following syntax to either commit or abort a transaction.

```
$session->commitTxn( );
```

or

```
$session->abortTxn( );
```

The changes made with a transaction are not visible outside of the script until the changes are committed. Within a transaction, the same script can see the proposed changes. Transactions also can control how other applications see objects before changes are committed or aborted by adding a single keyword. The syntax of a transaction with a keyword is:

```
$session->
   transaction(["WRITE_LOCK"|"READ_LOCK"|"NO_LOCK"]);
```

A keyword can be any one of the following:

Table 3-2 Transaction lock options

Keyword	Description
WRITE_LOCK	While the transaction is open, no other process can modify or access information in the repository.
READ_LOCK	Currently behaves the same as WRITE_LOCK
NO_LOCK	This is the default behavior. No locks exist until the script commits the transaction.

Transactions may be nested. When you nest a transaction, you must commit or abort the nested transaction before you commit or abort the previous transaction.

The API aborts any open transactions when the script terminates.

Example:

```
#! /usr/local/bin/Perl
$session = InCharge::session->init( );
$delthis = shift @ARGV;
$delthisObj = $session->object($delthis);
@relObj = @{ $delthisObj->{ComposedOf} };
$session->transaction();
$x = $delthisObj->delete();
foreach $mem (@relObj) {
    $mem->delete();
}
$session->commitTxn();
print("Deleted ".delthis." and related ports\n”);
```

In the example, the script deletes a card and its related ports. The script is invoked with an argument that specifies the card to delete. Using the ComposedOf relationship, the script creates a list of Port objects to delete. The script deletes the card and its related ports at the same time through a transaction ensuring that no other script can see the intermediate stage with an incompletely deleted suite of objects.

delete

```
$session->delete( $object );
```

Deletes the specified object instance from the repository.

Note: This does not clean up all the object inter-dependencies and links. For a cleaner object deletion, use the *remove()* operation, if one exists, for the object class in question; see the invoke() primitive.

The *delete()* method can be called in one of two ways.

```
$session->delete( $object );
```

or

```
$object->delete();
```

getEventType

```
$type = $session->getEventType( $class, $event );
```

Given a class and event name, this call returns a string that describes the type of the event. The possible strings returned are:

Table 3-3 Event types

Event type literal	Description
EVENT	Event
AGGREGATION	Aggregation
SYMPTOM	Symptom
PROBLEM	Problem
UNKNOWN	Error indication

Example:

```
$type = $session->getEventType( "Router", "Down" );
```

To obtain the low-level numeric type codes, instead of descriptive strings, use the *getEventType()* primitive, as shown.

```
$type =
   $session->primitive( "getEventType", "Router", "Down" );
```

getServerName

```
$session->getServerName( );
```

Returns the name of the Domain Manager to which the EMC Smarts session is connected.

insertElement

```
$session->insertElement( $object, $relation, @item[s] );
```

Inserts one or more elements into an object relationship.

It is suggested that the *insertElement()* feature of the *InCharge::object* module be used instead, as shown.

```
$obj->insertElement( $relation, @item[s] );
```

removeElement

```
$session-E>removeElement( $object, $relation, @item[s] );
```

Removes one or more elements from an object relationship, such as ComposedOf.

It is recommended that the *removeElement()* feature of the *InCharge::object* module be used instead, as shown.

```
$obj->removeElement( $relation, @item[s] );
```

4

Primitives

The Domain Manager primitives are low-level remote calls that are supported by the API. These primitives provide the standard protocol between client applications, such as dmctl, ASL adapters, API scripts, and the EMC Smarts Console, and the Domain Manager. This chapter describes the Perl syntax for calling these primitives and provides outline documentation on their use and semantics in the following sections:

- Primitive naming conventions 4-2
- Primitive calling conventions 4-3
- Error handling 4-4
- Error codes 4-5
- Data types 4-6
- Primitives 4-10

Primitive naming conventions

Module name

InCharge::primitives - Low-level primitive calls to Domain Managers

Conventions

The names given to the primitives follow a convention of using lower case, except for the first letter of the second and subsequent words of multi-word names. For example, to get operation arguments, we use the name getOperationArguments.

Where the resulting names are overly long, the API provides shorter aliases; getOperationArguments() has the alias getOpArgs(). Typically, the word “Operation” is shortened to “Op”, and “Property” is shortened to “Prop”, however, both the long and shortened name can be used. Both forms are listed here.

Since primitives are designed to be called via the *InCharge::session*, where a primitive name conflicts with a module function, the name of the primitive has the string “_P” concatenated onto it in order to differentiate the two. Script authors are discouraged from using these “_P” versions since higher-level versions are available through InCharge::session and, in some cases, InCharge::object that are easier to use.

Where a primitive returns a value that may be of any type, a second version of the call is provided that returns both the numeric type code and the return value. The name of this extended version is the same as the lesser original but with “_t” appended. You can also specify “_T” instead of “_t”, in which case when the primitive returns an ANYVAL_ARRAY_SET (that is, a structure of structures), the fields of the structures are also accompanied by their types. This is a reference to a two-element array containing type and value for each structure field.

The primitive names are similar to those used in the C++ API. Where the names do not match those used by ASL or dmctl, aliases are provided. For example, the ASL command *get* Instances() is called *get* Leaf Instances() in the C++ API, therefore, the API allows both names to be used. The C++ name is the name used for the actual primitive and the ASL name is provided as an alias.

The C interface for EMC Smarts, on the other hand, uses function names that look like sm_property_unsubscribe(). They start with “sm_” and use all lower-case words delimited by underscores. This set of functions is less complete than the C++ equivalent interface and does not provide a one-to-one match of all the Domain Manager primitives. The API for Perl does not provide a match for the C interface function names.

Primitive calling conventions

All the functions described in this document must be invoked with reference to a valid object of the InCharge::session module. These object references are created using InCharge::session->object(), InCharge::session->create(), or InCharge::session->getInstances().

The general approach used is as follows.

1. Initialize an EMC Smarts session and obtain a reference to it.

```
$session = InCharge::session->init( );
```

2. Call the primitives required using the session reference. For example:

```
foreach $class ( sort $session->getClasses() ) {
   foreach $inst (
      sort $session->getInstances($class))
   {
      print $class . "::" . $inst . "\n”;
   }
}
```

3. Close the session.

```
$session->detach( );
```

Where access to operations or properties of EMC Smarts repository objects is required you are discouraged from using the get(), put() and invokeOperation() primitives, but encouraged to use the features of the InCharge::object module instead. Using this approach, the script obtains an InCharge::object reference, which is used to access the required information. For example,

1. Establish an EMC Smarts session.

```
$session = InCharge::session->init();
```

2. Create an *InCharge::object* valid reference to the object of interest.

```
$obj = $session->object( "Router::gw1" );
```

3. Manipulate the object using the reference.

```
$type = $obj->{Type};
$obj->{Vendor} = "Cisco";
$fan1 = $obj->findFan( 1 );
```

4. Close the EMC Smarts session.

```
$session->detach( );
```

Error handling

All the functions and methods of objects in the API modules throw errors using the Perl *die* command. In order to catch any errors that may occur, the eval() function can be used and the "$@" variable inspected after the event. This is common Perl scripting practice.

The example shown in the following script will abort if the router "gw1" does not exist in the topology at the line where the name of the vendor is queried, and the last line will not be executed.

```
use InCharge::session;
$session = InCharge::session->init();
$vendor = $session
       ->object( "Router::gw1" )
       ->get("Vendor");
print "Vendor is $vendor\n";
```

To trap this possible error, the code can be modified as follows.

```
use InCharge::session; $session =
InCharge::session->init();
$vendor =
    eval{$session ->object( "Router::gw1" ) ->get(Vendor);
};
if ( $@ ) {
    print "Error obtaining the Vendor property\n";
} else {
    print "Vendor is $vendor\n";
}
```

For more details about using this mechanism, refer to the section on the eval and die functions in the Perl function man pages .

All error messages thrown by the API start with a number in square brackets. This is the error code and classifies the error as being one of those listed in Table 4-1. The remainder of the error text gives a verbose description of the specific error that was thrown. Where additional numeric codes are relevant, these are included in a second or subsequent set of square brackets.

The following example script attempts a connection with a Domain Manager and prompts for a username and password if the connection fails due to an authentication error: code 4.

```
my $domain = "SAM1";
my $user = undef;
my $passwd = undef;
for ( ; ; ) {
    $session = eval{ InCharge::session->new(
           domain => $domain,
           username => $user,
           password => $passwd);
    }
    if ( $@ =~ m/^\[4\]/ ) {
       print "Login: ";    chomp $user   = <STDIN>;
       print "Password: "; chomp $passwd = <STDIN>;
    } elsif ( $@ ) {
       die $@; # Some other fault
    } else {
       last;   # Success !
    }
}
```

Error codes

Table 4-1 Error codes

Error code	Error type	Description
1	Syntax error	Wrong number of arguments, missing argument, or too many arguments
2	System error	System call error; e.g., socket creation failed
3	Connection error	Socket connection error
4	Authentication error	Authentication error
5	HTTP error	Other session init failure (HTTP error in second number; e.g., “[5][301]”
6	Bad argument	Argument content or type error, or invalid name, invalid option, or wrong type; e.g., is scalar but reference wanted
7	Broker error	Cannot attach to broker
8	No domain	Domain not registered with broker
9	Protocol error	Protocol error, data size error, or unsupported protocol format
10	Isolated	Not attached
11	Invalid operation	Invalid or illegal operation
12	Bad function	Bad function call or primitive name
13	IO error	Socket IO error
14	Timeout	Timer expired
15	DM error	Error returned by Domain Manager
16	Not cached	Reply missing from cache
17	Configuration error	A required configuration element, such as an environment variable, is either missing or contains invalid data

Data types

The names of the variables used in the primitive descriptions below to denote the arguments and return values indicate the data type passed or expected. Although every effort has been made to use self-descriptive argument names in this guide, some need further explanation.

$session

This is a reference to a valid InCharge::session object - created using InCharge::session->new() or InCharge::session->init(). All Domain Manager primitives should be called with reference to an InCharge::session object, as shown:

```
$session = InCharge::session->init();
@list = $session->getClassInstances( "Router" );
```

$object

The specification of a repository object to be acted upon.

This can be given in one of the following formats:

Table 4-2 Formats to specify a repository object

Format	Description
'class::instance'	This format uses a single string, containing both the class and instance name with two colons between them. `$n = $session->countElements("Router::gw1", "ComposedOf");`
'::instance'	This format uses two parameters, where the first contains the class and the second contains the name of the instance. `$n = $session->countElements( "::gw1", "ComposedOf" );`
$class, $instance	This format uses two parameters, where the first contains the class and the second contains the name of the instance. `$n = $session->countElements( "Router", "gw1", "ComposedOf" );`
undef, $instance	This format uses two parameters, as above, but with the first replaced by the Perl *undef* value, indicating that it is unknown. This causes the API to perform a query to determine that name of the object's class. This syntax can only be used to refer to existing objects. `$n = $session->countElements(undef, "gw1", "ComposedOf");`

InCharge::object reference	This format is used whenever an object name is required. It is also possible to pass an *InCharge::object* reference.
	`$obj = $session->object( "Router::gw1" );`
	`$n = $session->countElements( $obj, "ComposedOf" );`

@objects

A list of objects is to be returned, which is only used as an return type.

The return is an array of object name strings in the format "ClassName::InstanceName".

$symptom, @symptoms

A number of calls return lists of symptoms. These are represented as an array of array references. Each subarray consists of four elements, each of which has the following significance:

```
$x[0] = type (INT)
$x[1] = certainly (FLOAT)
$x[2] = object (STRING - class::instance)
$x[3] = event/symptom name (STRING
```

You can gain access to the elements using the syntax,

```
$list[ $record_number ] -> [ $field_number ]
```

or,

```
$listref -> [ $record_number ] -> [ $field_number ]
```

The first of these is for use where the list is held in an array variable. The second is used when the list is held in an array pointed to by a reference.

$symptomData, @symptomData

Symptom data is returned as an array of nine values, as described below. When a list of symptoms is returned, it is formatted as an array of array references where each subarray contains the nine fields for a single symptom.

Table 4-3 Symptom data codes

Symptom data code	Description and type
$x[0]	state (INTEGER) 0 = active 1 = inactive 2 = suspended 3 = not monitored
$x[1]	last occurrence (LONG INTEGER)
$x[2]	instance display name (STRING)
$x[3]	class display name (STRING)
$x[4]	event type (INTEGER)
$x[5]	event certainty level (FLOAT)
$x[6]	event class (STRING)
$x[7]	event instance (STRING)
$x[8]	event name (STRING)

$type, @types

The Domain Manager protocol uses a range of integer values to identify the types of data being passed. These are used when a primitive is permitted to handle more than one data type as an argument or return value. For example, the invoke() primitive can take arguments of any type, such as integer, string, and Boolean. When specifying a type as a primitive function argument you can either use the numeric value or the mnemonic string, shown below. For a string, either use “13” or “STRING”. When type codes are returned by primitives, they are always returned as the numeric code.

To convert from the numeric code to the mnemonic string and back, use the built-in TYPE method of the InCharge::session module, as shown.

```
$mnemonic = $session->TYPE( $code )
```

or,

```
$code = $session->TYPE( $mnemonic );
```

The type code values used are:

Table 4-4 Type codes

Constant	Literal	Description
0	VOID	void (nothing)
1	ERR	error condition
2	BOOLEAN	boolean (1 = true, 0 = false)
3	INT	signed integer
4	UNSIGNED	unsigned integer
5	LONG	signed long integer
6	UNSIGNEDLONG	unsigned long integer
7	SHORT	signed short integer
8	UNSIGNEDSHORT	unsigned short integer
9	FLOAT	floating point
10	DOUBLE	double length floating point
12	CHAR	1-byte character
13	STRING	string
14	OBJREF	object (class and instance)
15	OBJCONSTREF	constant object reference
16	BOOLEAN_SET	set of booleans
17	INT_SET	set of signed integers
18	UNSIGNED_SET	set of unsigned integers
19	LONG_SET	set of signed long integers
20	UNSIGNEDLONG_SET	set of unsigned long integers
21	SHORT_SET	set of signed short integers

Table 4-4 Type codes (continued)

Constant	Literal	Description
22	UNSIGNEDSHORT_SET	set of unsigned short integers
23	FLOAT_SET	set of floating point numbers
24	DOUBLE_SET	set of double length floats
26	CHAR_SET	set of 1-byte characters
27	STRING_SET	set of strings
28	OBJREF_SET	set of objects (class and instance)
29	OBJCONSTREF_SET	set of constant object references
30	ANYVALARRAY	set of values (types included)
31	ANYVALARRAY_SET	two-dimensional array of values

$freshness

Where the function argument list takes a freshness parameter, this refers to how fresh the property being accessed by the function should be. This applies to polled or derived properties that may need recalculating or repolling if the property was last updated more than the specified $freshness seconds ago.

Primitives

classExists

```
$boolean = $session->classExists( $class )
```

Returns 1 if the specified class exists or 0 otherwise.

```
if ( $session->classExists( "Router" ) ) {
   print "Router class exists\n”;

}
```

consistencyUpdate

```
$session->consistencyUpdate( )
```

The *consistencyUpdate()* function causes the Domain Manager to recompute the correlation rules.

correlate

```
$session->correlate( )
```

Triggers the EMC Smarts ``Code book'' correlation actions, where symptoms are analyzed and correlated into problems.

countChildren

```
$count = $session->countChildren( $class )
```

Counts the child classes of the specified class.

```
$class = "ICIM_UnitaryComputerSystem";
$n = $session->countChildren($class);
```

countClassInstances

```
$count = $session->countClassInstances( $class )
```

Counts the number of objects that exist for a specified class, or those that would be returned by a call to *getClassInstances()*.

```
$n = $session->countClassInstances( "Router" );
```

countClasses

```
$count = $session->countClasses( )
```

Counts the number of classes present in the system.

```
$n = $session->countClasses( );
```

countElements

```
$count = $session->countElements( $object, $relation )
```

Counts the number of elements in the specified relationship.

```
$n = $session->countElements("Router::gw1","ComposedOf");
```

countInstances

```
$count = $session->countInstances( )
```

Counts the total number of objects in the repository, of all classes.

```
$n = $session->countInstances( );
```

countLeafInstances

```
$count = $session->countLeafInstances( $class )
```

Counts the number of leaf objects that exist for a specified class, those that would be returned by a call to *getLeafInstances()*.

```
$n = $session->countLeafInstances( "Router" );
```

countf

```
$count =
   $session->countf( $object, $relationship, $freshness )
```

Counts the number of elements in the specified relationship, such as countElements(). The contents of the relationship will be refreshed if the values are older than $freshness seconds.

createInstance

```
$session->createInstance( $object )
```

Note: Use the InCharge::session->create() function instead, as described in Chapter 3, "Session."

Creates a new ICIM object instance. The object specification must include both a class name and unique instance name.

```
$session->createInstance( "Router::fred" );
```

deleteInstance

```
$session->deleteInstance( $object )
```

Note: Use the InCharge::session->delete() or InCharge::object->delete() function instead, as described in Chapter 2, "Object" and Chapter 3, "Session."

Deletes the specified ICIM object instance from the database. Note that this does not clean up all the object inter-dependencies and links. For a cleaner object deletion, use the remove operation, if one exists, for the object class in question; see "invoke" on page 4-31 for more information.

```
$session->deleteInstance( "ACT_File::myFile" );
```

deleteObserver

```
$session->deleteObserver( )
```

Alias for purgeObserver().

Note: Consider this an internal call - use $session->detach() instead, as discussed in Chapter 3, "Session."

Reverses the effect of getObserverId(), deregistering the script as an observer.

```
$session->deleteObserver();
```

eventIsExported

Alias for getEventExported().

execute

Alias for executeProgram().

executeProgram

```
@thread = $session->executeProgram( $program, \@args )
```

Alias for execute().

Executes an EMC Smarts program, passing arguments to it.

The following example runs the dmdebug plug-in, displaying statistics information on the stdout file of the sm_server process.

```
@thread = $session->executeProgram (
       "dmdebug", [ "dmdebug", "--stats" ] );
```

exists

Alias for *instanceExists()*.

findInstances_P

Note: Use the findInstances() function from the InCharge::session module instead.

```
@objects = $session->findInstances_P(
       $class-pattern, $instance-pattern, $flag )
```

Finds instances that match the class and instance patterns, according to rules specified in the flags.

When used by the console GUI, the $flag value is 0x101000, which requests subclass expansion and glob pattern matches. When used by dmctl, the value 0x001000 is used which requests RegEx pattern matches and subclass expansion.

The value of $flag consists of the following values OR'd in any combination, according to the options required.

0x001000 = Expand-subclasses. With this flag set, the contents of subclasses of classes that match are also returned.

0x100000 = Glob. This causes the match to be done using ICIM glob() matches rather than UNIX regex syntax, which is used otherwise.

```
@list = $session->findInstances_P(
             "Router", "s*", 0x100000 );
@list = $session->findInstances_P(
    "ICIM_UnitaryComputerSystem", ".*", 0x001000);
```

forceNotify

```
forceNotify( $object, $event, $notified, $expires)
```

Notifies, or clears, the specified event.

The $notified and $expires parameters are both timers. If $notified is greater than or equal to $expires, then the event is cleared. If $notified is less-than $expires then the event is notified, or raised. The actual values of these parameters are not significant.

```
# to notify an event:
$session->forceNotify("Router::gw1",
           "Unresponsive", 0, 1);
 # to clear an event:
$session->forceNotify("Router::gw1",
           "Unresponsive", 0, 0);
```

get

```
RETURN = $session->get( $object, $property )
```

Note: Use the features of the InCharge::object module instead, as discussed in described in Chapter 2, "Object" and Chapter 3, "Session."

Gets the contents of the specified property of the object.

The return type is scalar, array, or array reference as appropriate.

```
$vendor = $session->get( "Router::gw1", "Vendor" );
@parts = $session->get( "Router::gw1", "ComposedOf" );
```

The preferred implementation is:

```
$object = $session->object( "Router::gw1" );
$vendor = $object->{Vendor};
@parts = $object->{ComposedOf};
```

get_t and get_T

```
($type, $value) = $session->get_t( $object, $property )
```

Like get(), this returns the contents of the specified property, however, get_t() also returns a code for the type of the data. The returned value will be a scalar or array reference, as appropriate.

```
( $type, $value ) = $session->get_t(
             "Router::gw1", "Vendor" );
( $type, $value ) = $session->get_t(
             "Router::gw1", "ComposedOf" );
```

The get_t() variant of this call also returns the types of values contained in complex structures. Where get_t() would return a value, get_T() returns a type code and value in a two-element array.

getAggregationEvents

```
@list = $session->getAggregationEvents( $object,
                    $eventname, $flag)
```

Gets the names of the events that are aggregated to the specified event, which must be an aggregation event type.

If $flag is false then the events directly aggregated are returned. If $flag is true, then the aggregation tree is walked, and the names of all nonaggregation events that the specified event ultimately depends on are returned.

```
@list = $session->getAggregationEvents(
       "Router::gw1", "PowerSupplyException", 1 );
```

getAllEventNames

```
@events = $session->getAllEventNames( $class )
```

Alias for getEvents().

Gets the list of all events of all types, including symptoms, problems, aggregates, and events, in no particular order.

The *getEventNames()* call is similar but omits the problems from the list.

```
@list = $session->getAllEventNames( "Router" );
```

getAllInstances

```
@instances = $session->getAllInstances( )
```

Gets the names of all instances present in the ICIM database.

Note: This potentially returns a very large array indeed and should, therefore, be avoided.

getAllProperties and getAllProperties_t

```
@properties = $session->getAllProperties( $object, $flag );
```

Returns the names and values of all the properties of the specified object.

If $flag is 0, attributes only are returned. If $flag is 1, relations only are returned. If $flag is 2, both attributes and relations are returned.

The @properties array contains an even number of elements, where the odd numbered ones are the property names and the even numbered are the matching values. This convention means that you can treat the result as a Perl hash, as shown in the example.

```
%props = $s->getAllProperties( $obj, 2 );
print "Object Name is $props{Name}\n”;
```

or,

```
use Data::Dumper;
print Dumper( \%props );
```

The "_t" variation of the call returns data types as well as values.

Consider using the *get()* or *get_t()* functions of the *InCharge::object* module with no arguments instead of this call, as shown.

```
%props = $obj->get( );
print Dumper( \%props );
```

getArgDirection

```
$direction = $session->getArgDirection( $class, $operation, $argname )
```

Gets a flag to indicate whether the specified operation argument is an IN or OUT argument.

IN arguments are denoted by the value 0 and refer to argument values passed from the script to the Domain Manager. OUT arguments are denoted by the value 1 and refer to variables into which the operation puts result information.

Nearly all arguments to all operations of all classes are IN arguments.

Note: OUT arguments are not supported by the remote access protocol; this discussion is beyond the scope of the API, dmctl, and ASL.

```
$direction = $session->getArgDirection(
             "Router", "getFan", "identifier" );
```

getArgType

```
$type = $session->getArgType( $class, $operation, $argname )
```

Alias for getOpArgType() and getOperationArgumentType().

Gets the type of the specified argument for the specified class operation.

Refer to the discussion about data types for a description of the possible values.

```
$type = $session->getArgType( "Router", "makeFan", "className" );
```

getAttributes

Alias for getAttributeNames().

getAttributeNames

```
@properties = $session->getAttributeNames( $class )
```

Alias for getAttributes().

Gets the list of all attributes for the specified class.

Attributes are properties that are not relations. For class Router, Vendor is an attribute but ComposedOf is not; however, both are properties. The getAttributeTypes() call returns the types of these attributes.

```
@list = $session->getAttributeNames( "Router" );
```

getAttributeTypes

```
@types = $session->getAttributeTypes( $class )
```

Gets the list of type codes associated with the attribute names returned by getAttributeNames().

The types returned by this call and the names returned by getAttributeNames() are in the same order, such that the type of $property[$n] is given in $type[$n]. Refer to the discussion about data types for a description of the possible values.

```
@list = $session->getAttributeTypes( "Router" );
```

getByKey

```
RESULT = $session->getByKey( $object, $table,
              [ $keytype, $keyvalue ] )
```

Gets the entry in the named table from the object, indexed by its key.

Tables are properties that can contain arrays of values.

```
@driver = $session->getByKey(
       "GA_CompoundDriver::Bridge-Generic-Driver",
       "drivers", [ "INT", 10 ] );
```

getByKey_t and getByKey_T

```
($type, $value) = $session->getByKey_t( $object, $table,
                 [ $keytype, $keyvalue ] )
```

Identical to getByKey() but returns a code for the type of the result as well.

```
($type, $data) = $session->getByKey_t(
       "GA_CompoundDriver::Bridge-Generic-Driver",
       "drivers", [ "INT", 10 ] );
```

getByKeyf

```
RESULT = $session->getByKeyf( $object, $table,
              [ $keytype, $keyvalue ], $freshness )
```

Identical to getByKey() but takes the “freshness” of the entry into account.

```
@driver = $session->getByKeyf(
       "GA_CompoundDriver::Bridge-Generic-Driver",
       "drivers", [ "INT", 10 ], 120 );
```

getByKeyf_t and getByKeyf_T

```
($type, $value) = $session->getByKeyf_t( $object, $table,
              [ $keytype, $keyvalue ], $freshness )
```

Identical to getByKey_t() but takes the “freshness” of the entry into account.

```
($type, $data) = $session->getByKeyf_t(
       "GA_CompoundDriver::Bridge-Generic-Driver",
       "drivers", [ "INT", 10 ], 120 );
```

getChildren

```
@classes = $session->getChildren( $class )
```

Gets the list of classes that are child classes of a specified one; i.e., classes derived from the base class.

```
$class = "ICIM_UnitaryComputerSystem";
@list = $session->getChildren($class);
```

getClassDescription

```
$text = $session->getClassDescription( $class )
```

Gets a textual description of the class.

The fixed string "no description available" is returned if the class programmer has not provided a description message for the class.

```
$description = $session->getClassDescription( "Router" );
```

getClassHierarchy

```
@hierarchy = $session->getClassHierarchy( );
```

Returns an array of information that provides a complete description of the hierarchy of domain model classes.

Each element of the array is a reference to a three-element subarray, the fields of which have the following significance.

Table 4-5 Class hierarchy descriptor

Array element	Description
$x[0]	name of ICIM class
$x[1]	name of the class's parent class
$x[2]	class is abstract flag: 1 = yes, 0 = no

getClassInstances

```
@instances = $session->getClassInstances( $class )
```

Gets the list of instances of a specified class.

The return is a list of strings that contain the instance names without the class name. For example, "fred" is returned rather than "Router::fred". This differs from getLeafInstances() in that this call returns the members of the class and any derived classes, whereas getLeafInstances() returns only the members of the specified class.

```
@names = $session->getClassInstances( "Router" );
```

getClasses

```
@classes = $session->getClasses( )
```

Gets the list of classes present in the system.

The following code fragment displays the list of all instances of all classes in the database.

```
foreach $class ($session->getClasses()) {
   foreach ($session->getClassInstances($class)) {
      print "${class}::$_\n";

   }
}
```

getCorrelationParameters

```
@info = $session->getCorrelationParameters( )
```

Returns a nine-element array, each element of which contains a parameter relating to the Domain Manager correlation mechanism.

The array elements have the following significance:

Table 4-6 getCorrelationParameters return values

Element	Description
info[0]	max problems (INT)
info[1]	correlation interval (INT)
info[2]	codebook radius (FLOAT)
info[3]	correlation radius (FLOAT)
info[4]	lost symptom probability (FLOAT)
info[5]	spurious symptom probability (FLOAT)
info[6]	time limit (INT)
info[7]	suspend correlation (BOOLEAN)
info[8]	provide explanation (BOOLEAN)

getEnumVals

```
@strings = $session->getEnumVals( $class, $property )
```

Returns the list of strings representing the possible values for an enumerated property.

This can be used to present a list of valid values to the user in the form of a selection menu. If this primitive is used to refer to an property that is not an enumerated one, and error is thrown.

```
@values = $session->getEnumVals( "Router", "Type" );
```

getEvents

Alias for *getAllEventNames()*

getEventCauses

```
@symptoms = $session->getEventCauses( $object, $eventname, $flag )
```

Gets a list of the Root causes, or problems, that the specified event can be considered to be a symptom of.

The getProblemClosure() primitive provides the reverse mapping. This is the mechanism used to populate the codebook tab for an event property sheet in the administrative console.

The $flag parameter is optional. If it is passed as TRUE, the full list of problems explaining eventname, whether directly or indirectly, is returned. If it is passed as FALSE, only those problems that directly list eventname among the events they explain are returned.

```
@causes = $session->getEventCauses(
       "Router::gw1", "MightBeUnavailable", 1 );
```

getEventClassName

```
$class = $session->getEventClassName( $class, $event )
```

The getEventClassName() function returns a string with the name of the ancestor class associated with a class and an event. The ancestor class is where the event was originally defined; i.e., the class in which the event definition statement, not any refinement, appeared.

```
$class = $session->getEventClassname( "Router", "Down" );
```

getEventDescription

```
$text = $session->getEventDescription( $class, $event )
```

The getEventDescription() function returns a string, defined in MODEL, that describes an event.

```
$descr = $session->getEventDescription("Router", "Down");
```

getEventExplainedBy

```
@symptoms = $session->getEventExplainedBy( $object, $event, $flag )
```

Note: Use *InCharge::session*->getExplainedBy instead, as discussed in Chapter 3, "Session."

Returns the list of symptoms that are explained by the specified impact event.

The $flag is a boolean that indicates whether the event impact tree is to be walked during the processing of the request.

```
@list = $session->getEventExplainedBy(
          "Router::gw1", "DownImpact", 1);
```

getEventExported

```
$boolean = $session->getEventExported( $class, $event )
```

Alias for *eventIsExported()*.

Returns 1 if the specified event is exported by the class or 0 if it is not exported. Events that are not exported are hidden from view in the GUI.

```
if ( $session->getEventExported( "Router", "Down" ) ) {
   print "Event is exported\n";
}
```

getEventNames

```
@events = $session->getEventNames( $class )
```

Gets the list of events handled by the specified class.

Some of the returned events are exported while others are not as described in getEventExported(). Unlike getAllEventNames(), this call does not return problems names.

```
@list = $session->getEventNames( $class );
```

getEventSymptoms

```
@events = $session->getEventSymptoms( $class, $event )
```

Returns the list of events that are symptoms of the specified one.

```
@symptoms =
    $session->getEventSymptoms( "Router", "Down" );
```

getEventType_P

```
$eventtype = $session->getEventType_P( $class, $event )
```

Note: Consider using InCharge::session->getEventType instead, as discussed in Chapter 3, "Session." The "_P" is included in the name in order to avoid confusion with the InCharge::session function of the same name.

This primitive returns a numeric code that indicates the type of the specified event. Possible values are shown in the table.

Table 4-7 getEventType return codes

Return code	Event type
0	Event
1	Aggregation
2	Symptom
3	Causality
4	Problem
5	Imported event
6	Propagated Aggregation
7	Propagated Symptom
8	Same type

```
$eventtype = $session->getEventType_P( "Router", "Down");
```

Note: The primitive version of this call must be called using the primitive function as shown because there is an *InCharge::session* method of the same name, provided for functional compatibility with ASL.

getInstances

Alias for getClassInstances().

getInstrumentationType

```
$type = $session->getInstrumentationType( $object )
```

Returns the instrumentation type for a specified object.

```
$type =
   $session->getInstrumentationType("Router::gw1");
```

getLeafInstances

```
@instances = $session->getLeafInstances( $class )
```

Alias for getInstances().

Gets the list of instances of a specified class.

The return is a list of strings that contain the instance names without the class name. For example, “fred” is returned rather than “Router::fred”. This differs from getClassInstances() in that this call returns only the members of the specified class, whereas the getClassInstances() call returns the members of the class and its derived classes.

```
@names = $session->getLeafInstances( "Router" );
```

getLibraries

```
@libs = $session->getLibraries( )
```

Alias for getModels().

Gets the list of libraries loaded into the system.

getModels

Alias for getLibraries().

getMultipleProperties and getMultipleProperties_t

Note: Use the get() and get_t() functions of the InCharge::object module with multiple arguments instead of this call, as shown in the example.

```
( $vendor, $model ) = $obj->get( "Vendor", "Model" );
```

The syntax of the primitive itself is,

```
@values =  $session->getMultipleProperties( $object, \@propnames);
```

or,

```
@values =  $session->getMultipleProperties_t($object, \@propnames);
```

For example,

```
($vendor,$model) =
   $session->getMultipleProperties( $obj,
                [ "Vendor", "Model" ] );
```

The argument is a reference to an array containing the names of the properties to be returned.

getObserverId

```
$id = $session->getObserverId()
```

Note: Do not call this primitive directly, but make use of the InCharge::session->observer() function instead.

Creates and returns a new observer id.

The deleteObserver() primitive reverses this action.

getOpArgType

Alias for *getArgType()*.

getOpArgs

```
@argnames = $session->getOpArgs( $class, $operation )
```

Alias for getOperationArguments().

Gets the names of the arguments for a specified class operation.

The argument names are returned in the order in which they should appear in the argument list when invoking the operation.

```
@list = $session->getOpArgs( "Router", "makeIP" );
```

getOpDescription

```
$text = $session->getOpDescription( $class, $operation )
```

Alias for getOperationDescription().

Returns a textual description of the specified class operation.

```
$description = $session->getOpDescription(
              "Router", "makeIP");
```

getOperationArguments

Alias for getOpArgs().

getOperationArgumentType

Alias for getArgType().

getOperationDescription

Alias for getOpDescription().

getOperationFlag

Alias for getOpFlag().

getOperationReturnType

Alias for getOpReturnType().

getOperations

Alias for getOpNames().

getOpFlag

```
$flag = $session->getOpFlag( $class, $operation )
```

Alias for getOperationFlag().

Gets the flag associated with the specified class operation.

The value returned is between 0 and 3, as defined in the table.

Table 4-8 getOpFlag return codes

Return code	Operation
0	No flag
1	Idempotent
2	Constant
3	Read only

```
$flag = $session->getOpFlag( "Router", "makeIP" );
```

getOpNames

```
@operations = $session->getOpNames( $class )
```

Alias for getOperations().

Gets the list of operations for the specified class.

The operations are returned as an array of strings containing their names.

```
@list = $session->getOpNames( "Router" );
```

getOpReturnType

```
$type = $session->getOpReturnType( $class, $operation )
```

Alias for getOperationReturnType().

Returns the return type code for the specified class operation.

Using this function you can determine whether the operation returns an integer, a string, an object, or a list. The type codes returned are integer numbers, as described for $type in notes about data types, above.

```
$type_code = $session->getOpReturnType(
                 "Router", "makeIP" );
```

getParentClass

```
$class = $session->getParentClass( $class )
```

Returns the name of the class from which the specified class is derived.

This is the logical inverse of getChildren().

```
$parent = $session->getParentClass( "Router" );
```

getProblemClosure

```
@symptoms = $session->getProblemClosure( $object, $eventname, $flag )
```

Note: Use *InCharge::session*->getClosure() in preference to this call.

Lists the events, or symptoms, that contribute to a specified problem. The *getEventCauses()* primitive is the inverse of this one.

```
@list = $session->getProblemClosure(
              "Router::gw1", "Down", 1 );
```

getProblemExplanation

```
@list = $session->getProblemExplanation( $object, $eventname, $flag )
```

Note: Use *InCharge::session*->getExplains() in preference to this call.

MODEL developers can add information to a problem in order to emphasize events that occur because of a problem. This function returns a list of these events.

```
@list = $session->getProblemExplanation( "Router::gw1",
                    "Down", 1 );
```

getProblemNames

```
@list = $session->getProblemNames( $class )
```

Get the event names of problems associated with the specified class.

```
@problems = $session->getProblemNames( "Router" );
```

getProblemSymptomState

```
@symptomData = $session->getProblemSymptonState( $object, $eventname )
```

Returns data about all the symptoms that indicate the specified problem, including significant state information.

```
@list = $session->getProblemSymptonState( "Router::gw1", "Down");
```

getPrograms

```
@list = $session->getPrograms( )
```

Gets the list of ``programs'' running in the Domain Manager.

Typically the reply list includes “dmboot” and “icf.”

```
@progs = $session->getPrograms( );
```

getPropAccess

```
$access = $session->getPropAccess( $class, $property )
```

Returns a number that indicates the level of access to the specified property.

This effectively identifies the method by which the property value is obtained internally. Possible values and their meanings are:

Table 4-9 getPropAccess return codes

Return code	Property access level
0	No access
1	Stored
2	Computed
3	Instrumented
4	Propagated
5	Uncomputable
6	Computed with expression

```
$access = $session->getPropAccess( "Router", "Vendor" );
```

getPropDescription

```
$text = $session->getPropDescription( $class, $property )
```

Alias for getPropertyDescription().

Returns a textual description of the named class property.

```
$descr =
   $session->getPropDescription( "Router", "Vendor" );
```

getProperties

Alias for getPropNames().

Note: The functionality of the C++ function getProperties() is available through the getMultipleProperties() primitive, and more easily through the get() method of the InCharge::object module.

Primitive getProperties() is aliased to getPropNames() in order to provide dmctl syntax compatibility.

getPropertyDescription

Alias for getPropDescription().

getProperties

Alias for getPropNames().

Note: For C++ developers, the C++ API call getProperties() is called getMultipleProperties(). However, the InCharge::object->get() is an easier way to use this functionality.

getPropertyType

Alias for *getPropType()*.

getPropIsReadonly

```
$boolean = $session->getPropIsReadonly( $class, $property )
```

Indicates whether the specified class property is read-only.

```
if ( $session->getPropIsReadonly( "Router", "Vendor" ) ) {
   print "Vendor is readonly\n”;
} else {
   print "Vendor can be changed\n”;
}
```

getPropIsRelationship

```
$boolean = $session->getPropIsRelationship( $class, $property )
```

Indicates whether the specified class property is a relationship.

```
if ( $session->getPropIsRelationship( "Router",
                  "ComposedOf" ))
{
   print "ComposedOf is a relationship\n”;
}
```

getPropIsRequired

```
$boolean = $session->getPropIsRequired( $class, $property )
```

Indicates whether the specified class property is required to have a value.

```
$needed =
   $session->getPropIsRequired( "Router", "Vendor" );
```

getPropNames

```
@list = $session->getPropNames( $class )
```

getPropRange

```
@range = $session->getPropRange( $class, $property )
```

Returns the range of valid values for the class property, provided the property has been defined.

This applies to a very limited number of properties of integer type, typically in polling configuration classes.

```
( $min, $max ) = $session->getPropRange(
          "DialOnDemand_Interface_Setting",
          "MaximumUptime" );
```

getPropType

```
$type = $seesion->getPropType( $class, $property )
```

Alias for getPropertyType().

Returns the data type for a specified class property.

"$type, @types" on page 4-8 describes the possible data types. This call always returns the integer number representation of the type.

```
$type = $session->getPropType( "Router", "Vendor" );
```

getPropertySubscriptionState

```
$state = $session->getPropertySubscriptionState(
                    $object, $property )
```

Gets the current state of subscription to the specified event.

The possible reply values are:

Table 4-10 getPropertySubscriptionState return codes

Return code	Subscription state
0	Unsubscribed
1	Pending
2	Subscribed
3	Suspended

getRelatedClass

```
$class = $session->getRelatedClass( $class, $property )
```

Returns the name of the class of object that can be related to the specified class through the property, which must be a relationship.

```
$class =
      $session->getRelatedClass( "Router", "ComposedOf" );
```

getRelationNames

```
@properties = $session->getRelationNames( $class )
```

Alias for getRelations().

Gets the names of all the relationship properties for the specified class.

```
@relationships = $session->getRelationNames( "Router" );
```

getRelations

Alias for *getRelationNames()*.

getRelationTypes

```
@types = $session->getRelationTypes( $class )
```

Returns a list of type numbers for the relationships which are returned by the getRelationNames() call.

```
@types = $session->getRelationTypes( "Router" );
```

"$type, @types" on page 4-8 describes the possible data types.

getReverseRelation

```
$property = $session->getReverseRelation( $class, $property )
```

Returns the name of the other end of a relationship pair denoted by the specified property name.

The inverse of ComposedOf is PartOf.

```
$relationship = $session->getReverseRelation(
                    "Router", "ComposedOf");
```

getSubscriptionState

```
$state = $session->getSubscriptionState( $object, $event )
```

Gets the current state of subscription to the specified event.

The possible values are:

Table 4-11 getSubscriptionState return codes

Return code	Subscription state
0	Unsubscribed
1	Pending
2	Subscribed
3	Suspended

getThreads

```
@list = $session->getThreads( )
```

Returns a list of threads running in the current Domain Manager system.

Each element of the returned array is a reference to a four-element array. The four values that describe each thread are, in order:

Table 4-12 getThreads return codes

Return array element	Thread information
$t[0]	Process ID
$t[1]	name
$t[2]	State
$t[3]	Status

This example prints the process IDs and names of all threads in process ID order.

```
foreach $t ( sort { $a->[0] <=> $b->[0] }
              $session->getThreads( ) ) {
   print $t->[0] . " - " . $t->[1] . "\n";
}
```

getf

```
RETURN = $session->getf( $object, $property, $freshness )
```

Gets the contents of the specified property of the object with reference to its freshness; see "Data types" on page 4-6. The return type is scalar, array, or array reference, as appropriate.

```
$vendor = $session->getf( "Router::gw1", "Vendor", 240 );
@parts =
   $session->getf( "Router::gw1", "ComposedOf", 360 );
```

getf_t and getf_T

```
($type, $value) = $session->getf_t( $object, $property, $freshness )
```

Like getf(), this returns the contents of the specified property but *getf_t()* also returns a code for the type of the data. The returned value will be a scalar or array reference, as appropriate.

```
( $type, $value ) =
      $session->getf_t( "Router::gw1", "Vendor", 240 );
( $type, $value ) =
      $session->getf_t("Router::gw1", "ComposedOf", 360);
```

getfAllProperties and getfAllProperties_t

```
%properties = $session->getfAllProperties($object, $flag, $freshness);
```

This is the same as getAllProperties(), but takes the freshness of the values into account and refreshes any stale properties before returning the results; i.e., those that are older than $freshness seconds.

See getAllProperties() for a description of $flag.

getfMultipleProperties and getfMultipleProperties_t

```
@values = $session->getfMultipleProperties( $object,
                    \@propNames,
                    $freshness );
```

Like getMultipleProperties(), but refreshes values that are staler than $freshness seconds and need re-polling.

The propNames argument must be a reference to an array of property names. For example,

```
@props = qw(Vendor Model Type);
($v,$m,$t) =
   $session->getfMultipleProperties($obj, \@props, 30);
```

hasRequiredProps

```
$boolean = $session->hasRequiredProps( $class )
```

Indicates whether or not the specified class has any properties that are flagged as required.

```
$reqd = $session->hasRequiredProps( "Router" );
```

insertElement_P

```
$session->insertElement_P($object,
            $relation,
            [ $type, $value ])
```

Note: Use the *InCharge::object::insertElement()* function instead, as described in discussed in Chapter 2, "Object." The "_P" on the end of the primitive name is used to avoid confusion between it and the *InCharge::session* and *InCharge::object* versions of the same call.

Inserts something into a relationshipset.

In order to access the low-level primitive version of this call, you must invoke it using the primitive method because the InCharge::session module also has its own variant.

```
$session->insertElement_P(,"Router", "ComposedOf",
              [ "OBJREF", "Fan::fan1" ]);
```

instanceExists

```
$boolean = $session->instanceExists( $object )
```

Alias for *exists()*.

Note: Use the ASL-like function InCharge::object::isNull() instead of this primitive. Note that the sense of the return value is reversed.

Indicates whether or not the named object is present in the repository.

However, the class name and instance name should be specified in the *$object* parameter.

```
$exists = $session->instanceExists( "Router::gw1" );
```

invoke

Alias for *invokeOperation()*.

invoke_t and invoke_T

Alias for *invokeOperation_t()*.

invokeOperation

```
RESULT = $session->invokeOperation($object, $operation, \@args)
```

Alias for *invoke()*.

Note: Use the features of the InCharge::object module instead, as described in Chapter 2, "Object."

Invokes a class operation on a specified object, passing the parameters to the operation.

The syntax of the arguments list requires it to be a reference to an array, each element of which is a reference to a two-element array containing the data type and value. Because of the awkward syntax, using the InCharge::object module provides a more natural style of interface.

```
$result = $session->invokeOperation(
          "Router::gw1", "makeInterface",
          [
          [ "INT", 1 ],
          [ "STRING", "interface-1" ],
          [ "STRING", "Interface" ]
          ] );
```

Note: The argument types, such as INT and STRING in the above example, can be specified using either their type names, as shown above, or numeric codes. "$type, @types" on page 4-8 describes the mapping.

invokeOperation_t and invokeOperation_T

```
( $type, $value ) =
   $session->invokeOperation_t($object, $operation, \@args)
```

Alias for invoke_t().

This is identical to invokeOperation() except that the return indicates the type of the returned data as well.

```
( $type, $value ) = $session->invokeOperation_t(
          "Router::gw1", "makeInterface",
          [
          [ "INT", 1 ],
          [ "STRING", "interface-1" ],
          [ "STRING", "Interface" ]
          ] );
```

The "_T" variation also embeds type codes into the fields of returned complex structures.

isAbstract

```
$boolean = $session->isAbstract( $class )
```

Indicates whether the specified class is abstract.

An abstract class is one from which other classes are derived but which cannot have any objects.

```
$class = "ICIM_UnitaryComputerSystem";
$flag = $session->isAbstract( $class );
```

isBaseOf

```
$boolean = $session->isBaseOf( $class1, $class2 )
```

Returns TRUE if $class2 is a base class of $class1; i.e., $class1 is derived from $class2.

Note: For the purposes of this query, all classes are taken to be derived from themselves.

```
$class1 = "Router";
$class2 = "ICIM_UnitaryComputerSystem";
$is_it = $session->isBaseOf( $class1, $class2 );
```

isBaseOfOrProxy

```
$boolean = $session->isBaseOfOrProxy( $class1, $class2 )
```

Returns TRUE (1) if $class2 is a base class or proxy class of $class1.

```
$class1 = "Router";
$class2 = "ICIM_UnitaryComputerSystem";
$is_it = $session->isBaseOfOrProxy( $class1, $class2 );
```

isInstrumented

```
$boolean = $session->isInstrumented( $class )
```

Indicates whether the specified class has associated instrumentation.

```
$flag = $session->isInstrumented( "TCPConnect" );
```

isMember

```
$boolean = $session->isMember( $object1, $relation, $object2)
```

Returns TRUE if $object2 is a member of the specified $object1 relationship.

```
$flag = $session->isMember( "Router::strrtbos",
             "ComposedOf",
             "Interface::IF-strrtbos/1" );
```

isMemberByKey

```
$boolean = $session->isMemberByKey( $object, $table,
                    $keytype, $keyvalue] )
```

Indicates whether an entry in the named object table exists.

```
$exists = $session->isMemeberByKey(
       "GA_CompoundDriver::Bridge-Generic-Driver",
       "drivers", [ "INT", 10 ] )
```

isMemberByKeyf

```
$boolean = $session->isMemberByKeyf( $object, $table,
             [$keytype, $keyvalue], $freshness)
```

Like isMemberByKey, but with reference to the freshness of the value.

```
$exists = $session->isMemeberByKeyf(
       "GA_CompoundDriver::Bridge-Generic-Driver",
       "drivers", [ "INT", 10 ], 120 )
```

isMemberf

```
$boolean = $session->isMemberf( $object1, $relation,
                $object2, $freshness )
```

Returns TRUE if $object2 is a member of the specified $object1 relationship. If the $relation is a computed or polled value and is more than $freshness seconds old, it is refreshed first.

```
$flag = $session->isMemberf( "Router::strrtbos",
              "ComposedOf",
              "Interface::IF-strrtbos/1",
              240 );
```

isSubscribed

```
$boolean = $session->isSubscribed( $object, $event )
```

Returns TRUE if the specified event has been subscribed to by the calling process.

```
$subscribed = $session->isSubscribed( "Router::gw1", "Down");
```

loadLibrary

```
$session->loadLibrary( $library )
```

Alias for loadModel().

Loads a library, model, into sm_server memory.

```
$session->loadLibrary( $libname );
```

loadModel

Alias for *loadLibrary()*.

loadProgram

```
$session->loadProgram( $program )
```

Loads the named program into sm_server memory.

```
$session->loadProgram( "dmdebug" );
```

noop

```
$session->noop()
```

Alias for ping().

This is a type of ping. It sends a null command string to the Domain Manager, thus determining whether the client/server link is active.

notify

See "notify" on page 2-7.

ping

Alias for noop().

propertySubscribe

```
$session->propertySubscribe( $object, $property, $interval )
```

Subscribes to notifications of changes to the specified object property.

See "Event subscription" on page 1-7. The actions of this call are reversed by propertyUnsubscribe().

```
$session->propertySubscribe( "Router::gw1",
              "Vendor", 30 );
```

propertySubscribeAll

```
$session->propertySubscribeAll( $flags,$class_pattern,
                $instance_pattern,
                $property_pattern, $interval);
```

Subscribes to changes in all the matching properties in the matching objects.

The meaning of the $flags is described for the subscribe() session function.

The actions of this call are reversed by propertyUnsubscribeAll().

```
$session->propertySubscribeAll( 0, "Router", "gw1",
                 ".*", 30 );
```

propertyUnsubscribe

```
$session->propertyUnsubscribe( $object, $property )
```

Reverses the effect of the propertySubscribe() call.

```
$session->propertyUnsubscribe( "Router::gw1", "Vendor" );
```

propertyUnsubscribeAll

```
$session->propertyUnsubscribeAll( $flags, $class_pattern,
                 $instance_pattern,
                 $property_pattern );
```

Unsubscribes from changes in all the matching properties in the matching objects.

The meaning of the $flags is described for the *subscribe()* session function.

purgeObserver

Alias for deleteObserver().

put_P

```
$session->put_P( $object, $property, [ $type, $value ] )
```

Writes the specified value to the specified object property.

This is the low-level primitive that the put() function of InCharge::session uses, and is called when using the hash dereferencing syntax of InCharge::object.

The reader is encouraged to use the InCharge::object logic.

The following are essentially equivalent.

```
$obj = $session->object( "Router::gw2" );
$obj->{Vendor} = "Cisco";
```

or,

```
$obj->put( "Vendor", "Cisco" );
```

or,

```
$ojb->put( Vendor => "Cisco", PrimaryOwnerContact => "Joe Blog" );
```

or,

```
$session->put( "Router::gw", "Vendor", "Cisco" );
```

or,

```
$session->put_P( "Router::gw", "Vendor", [ "STRING", "Cisco" ] );
```

quit

```
$session->quit( )
```

Alias for shutdown().

Closes down the Domain Manager cleanly, saving the configured parts of the repository to disk.

removeElement_P

```
$session->removeElement_P( $object, $relation, [ $type, $value ] )
```

Note: Use InCharge::object::removeElement() instead of this primitive. The "_P" in the name is there to save ambiguity between this primitive and the InCharge::session and InCharge::object equivalents.

This call removes an element from an object relationship, such as ComposedOf.

In order to access the low-level primitive version of this call, invoke it using the primitive method because the InCharge::session module has a method of the same name that provides an enhanced interface.

```
$session->removeElement_P("Router::gw", "ComposedOf",
           [ "OBJREG", "Host::pingu6" ] );
```

removeElementByKey

```
$session->removeElementByKey($object, $table, [$keytype, $keyvalue])
```

Remove a set-valued property by key.

restoreRepository

```
$session->restoreRepository( $filename, $purgeflag )
```

Restores the repository from file, optionally purging existing repository contents in the process.

```
$session->restoreRepository( "save.rps", 0 );
```

setCorrelationParameters

```
$session->setCorrelationParameters( @info )
```

Sets the Domain Manager correlation parameters.

For a description of the fields of the @info array, refer to the getCorrelationParameters() primitive. The example below sets the correlation interval to 20 seconds.

```
@info = $session->getCorrelationParameters( );
$info[1] = 20;
$session->setCorrelationParameters( @info );
```

shutdown

Alias for quit().

storeAllRepository

```
$session->storeAllRepository( $filename )
```

Note: Use the *InCharge::session->save()* method instead.

Saves the repository in the named file, which is located in the directory $SM_HOME/repos. The directory name must not contain any path separator characters.

```
$session->saveAllRepository( "save.rps" );
```

storeClassRepository

```
$session->storeClassRepository( $filename, $class )
```

Note: Use the InCharge::session->save() method instead.

Save the repository for the named class in the specified file.

```
$session->saveClassRepository( "save.rps", "Host" );
```

subscribeEvent

```
$session->subscribeEvent( $object, $event )
```

Subscribes to a specific event without using wildcard pattern matching, unlike subscribeAll().

The function unsubscribeEvent() cancels subscriptions established using subscribeEvent().

```
$session->subscribeEvent( "Router::gateway39", "Down" );
```

subscribeAll

Note: Use InCharge::session->subscribe() instead.

topologySubscribe

```
$session->topologySubscribe( )
```

Subscribes to notifications of topology updates.

The subscription/observer mechanism is described in detail in "Event subscription" on page 1-7. API subscriptions topology subscriptions may be reversed using topologyUnsubscribe().

topologyUnsubscribe

```
$session->topologyUnsubscribe( )
```

Cancels topology subscriptions previously requested with the topologySubscribe() function.

```
$session->topologyUnsubscribe( );
```

transactionAbort

```
$session->transactionAbort( )
```

Use *InCharge::session*>abortTxn instead.

Aborts a transactional block previously started using transactionStart().

transactionCommit

```
$session->transactionCommit( )
```

Note: Use *InCharge::session->commitTxn()* instead.

Commits a transactional block previously started using transactionStart().

transactionStart

```
$session->transactionStart( $lock_code )
```

Note: Use *InCharge::session->transaction()* instead.

Starts a transaction block, which may subsequently be aborted or committed using transactionAbort()/transactionCommit(). The $lock_code values have the following possible values:

Table 4-13 Lock code literals

Lock code	Literal
0	SM_NO_LOCK
1	SM_READ_LOCK_ONLY
2	SM_READ_LOCK
3	SM_WRITE_LOCK

```
sub SM_READ_LOCK { 1 };
$session->transactionStart( SM_READ_LOCK );
```

unsubscribeAll

Note: Use InCharge::session->unsubscribe() instead.

unsubscribeEvent

```
$session->unsubscribeEvent( $object, $event )
```

Note: Use InCharge:session::unsubscribe() instead.

Unsubscribes from the event previously subscribed using subscribeEvent().

Index

Symbols
$freshness 4-9
$object 4-6
$session 4-6
$symptom 4-7
$type 4-8
@objects 4-7
@symptoms 4-7
@types 4-8

A
abortTxn **3-17**, 4-37
API, C++ 4-26
API, script overview 1-4
ASL adapters 4-1

B
broken 3-7

C
C++ API 4-26
callPrimitive **3-10**, 3-12, 3-13
Certainty 1-9, 4-7
Class 1-2
classExists 4-10
clear 2-7
commitTxn **3-17**, 4-38
consistencyUpdate 4-10
correlate 4-10
countChildren 4-10
countClasses 4-10
countClassInstances 4-10
countElements **2-7**, 4-6, 4-10, 4-11
countf 4-11
countInstances 4-11
countLeafInstances 4-11
create 3-2, **3-9**, 4-11
createInstance 4-11

D
delete **2-7**, 3-18, **3-18**
deleteInstance 4-11
deleteObserver **4-12**, 4-35
detach 1-4, 1-5, 1-8, **3-7**, 4-3
Domain Manager 1-2, 1-3, 1-10

E
EMC Smarts
    Client 1-1, 1-7, 3-2
    Server 1-1, 1-7, 3-2
eval 3-4, 4-4
eventIsExported **4-12**, 4-19
execute 4-12
executeProgram 4-12
exists **4-12**, 4-30

F
findInstances 3-13
findInstances_P 4-12
forceNotify 4-13

G
get 1-3, 2-2, 2-3, **2-4**, 4-3, 4-4, 4-13, 4-15, 4-21, 4-26
get_T 4-13
get_t **2-4**, **4-13**, 4-15, 4-21
getAggregationEvents 4-14
getAllEventNames **4-14**, 4-18, 4-20
getAllInstances 4-14
getAllProperties **4-14**, **4-29**, 4-29, 4-30
getAllProperties_t 4-14
getArgDirection 4-15
getArgType **4-15**, 4-22, 4-23
getAttributeNames 4-15, **4-15**, 4-16
getAttributes 4-15
getAttributeTypes 4-15, **4-16**
getByKey **4-16**, 4-16
getByKey_t **4-16**, 4-16
getByKeyf 4-16
getByKeyf_T 4-16
getCauses 3-14
getChildren 1-3, **4-17**, 4-24

getClassDescription 4-17
getClasses 1-4, 3-3, 4-3, **4-17**
getClassInstances 1-9, 4-6, 4-10, **4-17**, 4-18, 4-21
getClosure **3-14**, 4-24
getCorrelationParameters **4-18**, 4-36
getEnumVals 4-18
getEventCauses **4-18**, 4-24
getEventClassName 4-19
getEventDescription 4-19
getEventExplainedBy 4-19
getEventExported 4-12, **4-19**, 4-20
getEventNames 4-14, **4-20**
getEvents 4-14, **4-18**
getEventSymptoms 4-20
getEventType **3-18**, 4-20
getEventType_P 4-20
getExplainedBy 1-3, **3-15**, 4-19
getExplains **3-15**, 3-15, 4-24
getf 4-29
getf_T 4-29
getf_t 4-29
getfAllProperties_t 4-29
getFileno 3-11
getfMultipleProperties 4-30
getfMultipleProperties_t 4-30
getInstances 1-3, 1-6, 3-3, 3-10, 4-2, 4-3, **4-21**
getInstrumentationType 4-21
getLeafInstances 4-2, 4-11, 4-17, **4-21**
getLibraries **4-21**, 4-21
getMultipleProperties 3-11, **4-21**, 4-26, 4-30
getMultipleProperties_t 4-21
getObserverId 4-12, **4-22**
getOpArgs 4-2, **4-22**
getOpArgType 4-15, **4-22**
getOpDescription 4-22
getOperationArguments 4-2, **4-22**
getOperationArgumentType 4-15, **4-23**
getOperationDescription 4-23
getOperationFlag **4-23**, 4-23
getOperationReturnType **4-23**, 4-24
getOperations 4-23
getOpFlag 4-23, **4-23**
getOpNames 4-23, **4-23**
getOpReturnType 4-23, **4-24**
getParentClass 4-24
getProblemClosure 4-19, **4-24**
getProblemExplanation 4-24
getProblemNames 4-24
getProblemSymptomState 4-25
getPrograms 4-25
getPropAccess 4-25
getPropDescription 4-25
getProperties 4-26
getPropertyDescription 4-25, **4-26**
getPropertySubscriptionState 4-27
getPropertyType **4-26**, 4-27
getPropIsReadonly 4-26
getPropIsRelationship 4-26
getPropIsRequired 4-26
getPropNames 4-26, **4-27**
getPropRange 4-27
getPropType 4-26, **4-27**
getProtocolVersion 3-11
getRelatedClass 4-27
getRelationNames **4-28**, 4-28
getRelations 4-28
getRelationTypes 4-28
getReverseRelation 4-28
getServerName 3-19
getSubscriptionState 1-8, **4-28**
getThreads 4-29

H

hasRequiredProps 4-30

I

init 1-4, 1-7, 1-9, 3-2, **3-6**, 3-18, 4-3, 4-4, 4-6
insertElement **2-6**, 3-19, 4-30
insertElement_P 4-30
instanceExists 4-12, **4-30**
invoke 1-3, 2-2, 2-3, **2-6**, 3-12, 3-13, 3-18, 4-8, 4-11, 4-31
invoke_T 4-31
invoke_t **2-6**, 3-13, 4-31
invokeOperation 3-13, 4-3, **4-31**, 4-31
invokeOperation_T 4-31
invokeOperation_t 4-31, **4-31**
isAbstract 4-32
isBaseOf 4-32
isBaseOfOrProxy 4-32
isInstrumented 4-32
isMember 4-32
isMemberByKey 4-33
isMemberByKeyf 4-33
isMemberf 4-33
isNull **2-6**, 4-30
IsSubscribed 1-8
isSubscribed **4-33**

L

loadLibrary 4-33
loadModel 4-33
loadProgram 4-34

M

Model 1-7

N

new 1-4, 1-7, 1-8, 2-3, 3-2, **3-5**, 3-6, 3-7, 4-4, 4-6
noop 4-34
Notification 1-9
notify **2-7**, 4-34

O

object 2-2
observer 1-7, 1-8, 1-9, **3-8**, 3-8, 4-22
Operation 4-5

P

Perl 1-3, 1-4, 1-7, 2-4, 2-6, 3-4, 3-18, 4-2, 4-4
ping 4-34
Primitive 1-3, 3-3, 3-12, 4-2, 4-5, 4-6, 4-8, 4-18, 4-22, 4-24, 4-30, 4-35, 4-36
primitiveIsAvailable 3-11
Property 1-2, 1-4, 1-8, 1-11, 1-12, 2-4, 2-6, 3-12, 4-13, 4-14, 4-15, 4-19, 4-25, 4-27, 4-28, 4-29, 4-34
propertySubscribe 1-7, 1-9, 2-3, **4-34**, 4-34
propertySubscribeAll 4-34
propertyUnsubscribe 1-8, 4-34, **4-34**
propertyUnsubscribeAll 4-34, **4-35**
purgeObserver 4-12, **4-35**
put 2-2, 2-3, **2-5**, 3-10, 3-12, 4-3, 4-35
put_P 3-12, **4-35**

Q

quit **4-35**, 4-36

R

reattach **3-7**, 3-7, 3-8
receiveEvent 1-7, 1-9, 1-13, 3-8, **3-8**
Relationship 4-11, 4-27, 4-28
removeElement 2-6, **2-7**, 3-19, 4-36
removeElement_P 4-35
removeElementByKey 4-36
restoreRepository 4-36

S

save **3-12**, 4-36, 4-37
setCorrelationParameters 4-36
shutdown 4-35, **4-36**
SM_AUTHORITY 3-5, 3-6
SM_HOME 3-6, 4-36
storeAllRepository 3-3, **4-36**
storeClassRepository 4-37
subscribe 1-8, 2-3, **3-16**, 3-17, 4-34, 4-35, 4-37
subscribeAll 1-8, 4-37
subscribeEvent **4-37**, 4-38
Subscription 1-7, 1-12, 3-7, 3-8, 4-27, 4-37

T

Timestamp 1-10, 1-11, 1-12
topologySubscribe 1-8, **4-37**
topologyUnsubscribe 1-8, **4-37**
transaction **3-17**, 4-38
transactionAbort **4-37**, 4-38
transactionCommit **4-38**, 4-38
transactionStart 4-37, 4-38, **4-38**

U

unsubscribe 1-8, **3-16**, 4-38
unsubscribeAll 1-8, **4-38**
unsubscribeEvent 4-37, **4-38**