VMware Smart Assurance Perl Reference Guide

VMware Smart Assurance 10.0.0

vmware®

You can find the most up-to-date technical documentation on the VMware website at:

https://docs.vmware.com/

If you have comments about this documentation, submit your feedback to

docfeedback@vmware.com

VMware, Inc.
3401 Hillview Ave.
Palo Alto, CA 94304
www.vmware.com

Copyright © 2019 VMware, Inc. All rights reserved. Copyright and trademark information.

Contents

1 Introduction 4

Fundamental concepts 4

Overview of a simple API Perl script 8

Using primitives and object references 9

Event subscription 10

2 InCharge::Object 17

Functions and methods 17

Overview 22

3 InCharge::Session 26

Overview 26

Function groups 27

Error handling 28

Session management functions 28

Utility functions 36

Specifying the locale while connected 38

Retrieving and setting log, error and trace levels at runtime 38

Wrapper functions 40

4 Primitives 50

Primitive naming conventions 50

Primitive calling conventions 51

Error handling 52

Error codes 53

Data types 54

Primitives 57

5 IPv6 Considerations 95

Conventions for specifying IPv6 addresses 95

Controlling name resolution 95

Introduction

1

This chapter includes the following topics:

- Fundamental concepts
- Overview of a simple API Perl script
- Using primitives and object references
- Event subscription

Fundamental concepts

The VMware Smart Assurance Remote Application Programming Interface (API) for Perl allows developers to create Perl scripts that connect to Domain Managers as clients to exchange information, manipulate data, or drive Domain Manager actions. The API provides access to all Domain Manager features, by 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. The API runs on UNIX platforms that support Perl 5.6.1 and Perl 5.8.x.

When using the provided Perl 5.8.8 version, the API uses a Flow module, which replaces IO::Socket and enables encryption and “keep alive” functionality. If you use Perl 5.6.1 there is no encryption or keep alive functionality.

The API also supports IPv4 and IPv6 environments.

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

A script developed using the Perl API can crete, delete and interact with instances of interfaces in a Domain Manager. The interfaces are defined using the MODEL language, compiled and loaded into a Domain Manager. 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 a Domain Manager. For example, Router is the name of a class, and all routers that are managed by a 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 a VMware Smart Assurance 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 VMware Smart Assurance software: attributes and relationships.

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.

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, you can 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, by passing information to them through arguments, and by obtaining the results of the action.

Events

Events describe the failures that can occur for a class, the symptoms that these failures create, and the effect of such 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 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.

The VMware Smart Assurance ASL Reference Guide provides further information about the VMware Smart Assurance data structures.

The Perl API is implemented as a set of Perl modules, which individual Perl scripts may access, by using the familiar ”use” directive. InCharge::session and InCharge::object offer the principal interface, which respectively provide 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.

Encryption

Messaging between the client and Domain Manager can be encrypted, by using the Perl API. Encryption is dependent upon the values set for the environment variables SM_INCOMING_PROTOCOL and SM_OUTGOING_PROTOCOL. If these values are not specified, encryption is automatically negotiated between the client and Domain Manager. The VMware Smart Assurance System Administration Guide provides further information about encryption.

“Keep Alive” messaging

The Perl API also supports “Keep Alive” messaging to maintain active connections between the client and DM, as discussed in the VMware Smart Assurance System Administration Guide.

Transcoding character data

You must transcode character data from your code page to UTF-8 before placing the character data into the remote API buffer. Data will be returned to your Remote API client application bindings as UTF-8 strings.

Setting environment variables

You must define the following variables to run a Perl script within the VMware Smart Assurance environment from the command line.:

- SM_HOME
- SM_WRITABLE
- SM_AUTHORITY
- SM_BROKER
- SM_BROKER_DEFAULT
- SM_SITEMOD
- SM_INCOMING_PROTOCOL
- SM_OUTGOING_PROTOCOL

  You can find further information on these variables in the VMware Smart Assurance System Administration Guide.

  To set these variables, type the following command to start the bash environment:

```
                   BASEDIR/smarts
                /bin/runcmd bash
bash$>
```

  A Perl script to define the variables can now be run within this bash environment.

“Keep Alive” and encryption requirements

The Perl API now supports “Keep Alive” and encrypted communications when run with the version of Perl supplied by VMware. You should use the sm_perl command that is shipped with the VMware Smart Assurance software, in order to successfully run the Perl API.

You can use a pure Perl implementation without keepalive or encryption by setting the environment variable SM_DISABLE_FLOW_WRAPPER.

Overview of a simple API Perl script

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

1 Open a session.

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

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

2 Work with the domain.

Call the primitives required, by 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.

4 Establish a session.

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

5 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, by using the object() method of the session handle.

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

6 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, by using Perl's hashing syntax and operations can be invoked, by using Perl's object-oriented syntax conventions.

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

7 Close the 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 Domain Managers. Each of these primitives can be invoked with reference to the *InCharge::session* handle, described in step 1 on page 19, 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, by using a syntax that is very similar to ASL. For example, in ASL you can list the vendors of all routers, by 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 in the ASL and Perl API example 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 Domain Managers.

The VMware Smart Assurance programming model delivers two different modes of client/server communication. The most direct is where a client makes a request of the 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 through subscriptions and is used when the client program needs to listen for events generated by the 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 a Domain Manager to send subscribed events to a client program, the client must first register itself with the Domain Manager as an event observer.

In Perl, this is done by using the observer() method of the InCharge::session module. Two steps are required:

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

First, the client must connect to the Domain Manager which establishe s a new InCharge::session connection. This is done by using either the InCharge::session->new() or InCharge::session->init() methods.

Here are some example code fragments that achieve this goal:

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

2 Attach an observer to the Domain Manager session.

Once the script has obtained a handle that references the script/server connection ($session in step 1 on page 22), it can be used to obtain a second connection to the notification engine of the Domain Manager. This is obtained by using the observer() method on the InCharge::session handle just obtained. The following code performs this action.

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

Subscribing to notifications

Once the client has registered itself as an observer, the next step is to inform the Domain Manager about which events the observer wants to receive notifications. VMware Smart Assurance allows clients to subscribe to a number of different types of events. These are listed in Subscription methods summary.

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, 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 timeout 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 by using the Perl syntax: $event[0], contains the event's timestamp measured by using normal UNIX time_t semantics (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 shown in Notification record - NOTIFY.

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, as shown in Notification record - CERTAINTY_CHANGE.

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)

Notification record - CLEAR describes the format of the record when an event is cleared by the Domain Manager.

Table 1-4. Notification record - CLEAR

Event record entry	Description
$event[0]	Timestamp (INTEGER)
$event[1]	“CLEAR”
$event[2]	Object class name (STRING)

Table 1-4. Notification record - CLEAR (Continued)

Event record entry	Description
$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. Notification record - CREATE describes the format of an object create record.

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. Notification record - DELETE describes the format of an object delete record.

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. Notification record - CLASS_LOAD describes the format of a class load record.

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. Notification record - RELATION_CHANGE shows the format of a relation change record.

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 shown in Notification record - ATTR_CHANGE.

Table 1-9. Notification record - ATTR_CHANGE

Event record entry	Description
$event[0]	Timestamp (INTEGER)
$event[1]	“ATTR_CHANGE”
$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. The observer describes the proper handling of DISCONNECT events if the restartableServer operation is appropriate. These records are generated even if no subscriptions to the Domain Manager are issued. The format of the Domain Manager disconnect message is shown in Notification record - DISCONNECT.

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 reconnection. The CONNECT message is an indication of success. The Perl API instead gives an immediate error on failure of the InCharge::session-->init() method 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, the Domain Manager does not report an error by using normal Perl *die* semantics. Instead, a notification is used to report that the subscription was rejected. The format of the ACCEPT/REJECT message is shown in Notification record - ACCEPT, REJECT, (PROPERTY).

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 shown in Notification record - SUSPEND (PROPERTY).

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)

Timeout records

If no event arrives within the time specified as the optional argument to receiveEvent(), a TIMEOUT record is returned, whose message format is shown in Notification record - TIMEOUT.

Table 1-13. Notification record - TIMEOUT

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

InCharge::Object

2

This chapter includes the following topics:

- Functions and methods
- Overview

Functions and methods

object

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

The object() method creates a new object reference.

get

The get() method is used to retrieve the value of the specified properties of the object, such as the value of an attribute. In addition, the call get_t also extends the functionality of the get() method by returning additional information that identifies the data type (for example, STRING) of the property value.

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 an error even when some properties cannot be retrieved.

Single property

You can access the content of a property whose name is held in a variable by using the Perl typical logic, as shown:

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

In this example, the value of the $propname variable is retrieved:

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

Multiple properties

You can also get multiple values in a single get() call by listing all the property names as arguments, by using the following syntax:

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

The results are returned in an array. This is faster than using multiple single-property get() calls. In this example, the value of the Vendor and Type attributes are retrieved:

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

All properties and relationships

You can also call get() with no arguments, in which case, a hash is created that contains all the object properties and relations, as shown:

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

There is no syntactical advantage, but there is a significant speed advantage.

Return value types

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 Return type for call syntax and Perl evaluation context.

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

For example, if the attribute MyValue is declared as an integer in the MODEL code, then the return type for that property will be an integer. Multiple values are always returned in an array or array reference.

get_t

The get_t() method is like the get() method, except that it returns the type of 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. The $session->TYPE() method can be used to convert the $type integer value to a mnemonic string.

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

There is a second syntax that gets the types and values for multiple properties. Each type/value pair is held in a two-element subarray within the returned data.

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

There is also a 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.

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

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 );
```

The put() method allows fields of the object to be modified in the Domain Manager 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 Domain Manager native ASL language, as shown:

```
$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");
```

You can also set more than one property in a single call, by using the following syntax:

```
%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();
```

The isNull() method 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 ... );
```

The invoke() method calls 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" );
```

You can also use the invoke_t() method, described in invoke_t to return additional information.

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, and from any relationships it belongs to. The delete() function does not remove any objects that had a “contains” type of relationship with the object being deleted. For example, calling delete() on a Router instance would remove that instance from the repository and remove that instance from any relationships that it was part of. However, the delete() function would not remove the Card objects to which the Router instance shares a “ComposedOf” relationship.

Consider using the remove() operation, if one exists, instead for a more complete action. For example, calling remove() on a Router will cause the Card objects it is composed of to be removed (which in turn could cause the Ports/Interfaces to be removed, and so on).

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. The countElements method will throw an error if $relation is not a relationship.

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

Overview

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

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

Name

InCharge::object

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 Domain Manager 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::sessionmodule. Both methods allow access to preexisting 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" );
```

If you do not 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 the Domain Manager 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, which imposes 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 by 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 by using either the get() and put() methods or the psuedo-hash syntax. The latter syntax is preferred because it is closer to the original built-in ASL language logic.

Two special internal properties can be accessed by 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 by 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 calls the new()operation of the object in the repository.
- 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 make other ICIM calls that refer to instances, such as subscribe(), use the features of InCharge::session directly. The following line of code is invalid:

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

  Instead, use one of the following:

- The first alternative:

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

- The second alternative:

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

The reason you have to use one of these two alternative methods is because the propertySubscribe is not a repository class operation, but rather it is a primitive.

```
dmctl -s <domain> getOperations <classname> | more
```

Likewise, to determine what properties can be accessed by using the InCharge::object module use:

```
dmctl -s <domain> getProperties <classname>
```

InCharge::Session

3

This chapter includes the following topics:

- Overview
- Function groups
- Error handling
- Session management functions
- Utility functions
- Specifying the locale while connected
- Retrieving and setting log, error and trace levels at runtime
- Wrapper functions

Overview

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

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

Name

InCharge::session

Synopsis

```
use InCharge::session;
```

There are three different syntaxes that you can can then choose from to connect to the Domain Manager:

- $session = InCharge::session->init( );
- $session = InCharge::session->new("INCHARGE");
- $session = InCharge::session->new(

  broker=>"localhost:426",

  domain=>"INCHARGE",

  username=>"noddy",

  password=>"bigears",

  traceServer => 1

  locale=>”en_US”

  );

```
 $object = $session->object( "Host::toytown1" );
 $object = $session->create( "Router::crossroads" );
(... and continuing with the methods described in the following sections...)
```

Description

This module provides the mechanisms for accessing a Domain Manager in a manner that is similar to that employed by the ASL language. It provides the main access point to 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.

The locale argument to the InCharge::session module allows you to set the locale of the session (the client locale).

Note You can use the InCharge::session->setLocale() method to change the locale for the session, while it is connected.

The default locale to set for a Perl client is determined by looking in the following places.

1. The value of the SM_LOCALE environment variable.
2. The default isen_US.

   Chapter 1, Introduction, provides 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 five kinds of functions: session management, Domain Manager primitives, utility, wrapper, and locale specification.

Session management functions

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

Domain Manager primitive functions

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 Perl scripts to be used with the SDK software.

Wrapper functions

This group of functions provides wrappers around the primitives to provide an interface that is more consistent with the SDK 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, described in Domain Manager primitive functions, is one such function.

Specifying the client locale

The InCharge::session module provides a function that enables you to set the locale of the session (the client locale).

Error handling

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

Session management functions

The following session management functions are provided.

new

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

The new function first establishes a connection between the calling Perl script and a Domain Manager. It then 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 are required if connecting to a server 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.

Option to specify the Broker

```
broker => $host[:$port]
```

This option specifies the 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 section Conventions for specifying IPv6 addresses describes how to specify an IPv6 address.

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

Option to specify the domain

This option 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.

There are two different syntaxes that can be used to specify the domain:

- domain => [$host:$port/]$domain
- server => [$host:$port/]$domain

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

Option to specify the username

This option 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, as described in Option to specify the password. 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.

There are two different syntaxes that can be used to specify the username:

- user => $user_name

- username => $user_name

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

Option to specify the password

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

```
password => $password
```

Option to specify a description of the script

This option 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.

```
description => $description
```

Option for specifying server-level tracing

If this option is 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.

Note When server-level tracing is turned on, it results in a large amount of data written to 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.

```
traceServer => 1
```

Option for specifying response timeout

This option specifies the timeout to be enforced while waiting for responses from the Domain Manager to primitive requests. The default value is 120 seconds. You may increase the value, if necessary, but do not set it to a value below 120 seconds. Otherwise, slow to-process requests will fail in a manner that looks like a communication link failure between the script and the Domain Manager.

```
timeout => $timeout
```

Option for specifying the session locale

The locale option allows you to set the locale of the session (the client locale).

```
locale=>locale
```

init

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

This funtion 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, trace, timeout, and locale 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
--locale=<locale>
```

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. Comments are included in the file for detailing this mechanism. This mechanism is turned on by specifying the value <STD> for the SM_AUTHORITY environment variable.

If the InCharge::session->init() functions encounters a command line syntax error, it calls usageError, in the main script, which the developer must provide. A single large text string that contains a description of the standard options handled is passed as the argument to usageError, which enables 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( );
```

The locale argument to the InCharge::session init method allows you to set the locale of the session (the client locale).

Note You can use the InCharge::session->setLocale() method to change the locale for the session, while it is connected.

The default locale to set for a Perl client is determined as follows:

1 The value of the SM_LOCALE environment variable.

2 If the SM_LOCALE variable is not set, then the default isen_US.

broken

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

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

A return value of non-zero indicates connection or protocol failures. To continue working with a broken session, the script should call the reattach() function, and then reestablish the event subscription profiles required.

reattach

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

Reestablishes a connection that has been detached or broken.

The reattach() function can be called to reconnect to a server to which the connection has been lost. Reestablishing the connection does not automatically reestablish 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, leaving the session in a working state. 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 describes error prefixes, including the [13] prefix.

Note The reattach() primitive does not return a new session identifier, but does refresh the referenced one. This is not a dup() style of action.

detach

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

The detach() function enables you to detach from the domain referred to by $session.

This function can be used for either a 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 reestablished. Thus, it is possible to call $session->reattach() to reconnect to the Domain Manager by using the same parameters as were used in the initial connection. However, the event subscriptions need to be reestablished explicitly in this event.

observer

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

The observer() function creates and returns a reference to a connection to a Domain Manager on which subscribed events can be received.

This establishes a new socket between the client and Domain Manager. Once connected, events can be subscribed to by using the various subscribe methods, and they can be received by 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 reconnection is not performed automatically. The script can use the $session->reattach() call to attempt an explicit reconnection and must then reestablish 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 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 22* 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 styles shown in the following examples:

- object('Router::gw1')

  In this example, the $objectname parameter is 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.

- object('Router', 'gw1')

  In this example, the $objectname parameter is specified as two strings with one for the class and one for the instance name.

- object('::gw1')

  In this example, the $objectname parameter is specified as 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 which causes a minor performance penalty.

- object(undef, 'gw1')

  In this example, the $objectname parameter is specified as two parameters with the first one undefined. This also results in the API performing a Domain Manager query.

- object('gw1')

  In this example, the $objectname parameter is specified as a single parameter that does not include the double-colon (::) delimiter, which must contain just the instance name. 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 );
```

Similar to the object() call, described in object, the create() call creates an InCharge::object valid reference through which a specified instance can be manipulated. However, unlike object(), the create() method creates the object if it does not 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.

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 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 the primitive 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. The put provides further details.

The type of the RESULT in array or scalar context is dependant on the primitive that is 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

The following utility functions are provided.

TYPE

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

The TYPE() function converts a Domain Manager data type symbolic name to its internal numeric code, or converts an internal numeric code to its symbolic name. So the following prints “13”:

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

The following code prints “STRING”:

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

getFileno

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

The getFileno( ) function returns a number that refers to the socket used for the script/server connection.

Note Do not use this function with the Perl **select** statement to listen for events from multiple domains by using multiple observer objects. Instead, use the new select function: InCharge::session::select.

getProtocolVersion

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

The getProtocolVersion( ) function 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 )
```

The primitiveIsAvailable() function 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 it 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};
}
```

select

@ret = InCharge::session::select(\@observerList, $timeout);

The select() function checks if there are data to be read for each observer in @observerList.

@ret returns a list of handles in which the value:

- 1 represents that the observer has data to read
- 0 indicates that there are no data to be read.

  The $timeout parameter indicates how many seconds the function select should wait before returning the result. It can be called with an undef value for no wait time.

  Example of usage:

  @ret = InCharge::session::select(<ref_list>,<timeout>);

  where:

- *<ref_list>* is a reference to the list of observers that will be checked.
- *<timeout>* is the number of seconds to wait before returning. For immediate return, use the argument *undef*.

  The following script returns an array with the result for each checked observer.

```
use InCharge::session;
use Data::Dumper;
my %common = ( user => "admin", password => "changeme");
$mask = "";
$i=0;
foreach $dom ( "INCHARGE-AM", "INCHARGE-SA" ) {
```

```
  $sess{$dom} = InCharge::session->new( %common, domain => $dom );
  $obs{$dom} = $sess{$dom}->observer( );
  @obsList[$i]= $obs{$dom};
        $sess{$dom}->subscribe( ".*::.*::.*/pa" );
  #$fn{$dom} = $obs{$dom}->getFileno()."\n";
  #vec($mask, $fn{$dom}, 1) = 1;
  $i++;
 }
 $i=0;
 for ( ; ; ) {
        $i=0;
  @obsRet = InCharge::session::select( \@obsList, undef);
  foreach $obsR ( @obsRet ) {
   #next unless (vec($rout, $fn{$dom}, 1));
   @event = $obsR->receiveEvent( );
   print "$obsR->{domain} - ".join( ", ", @event ). "\n";
   $i++;
  }
}
```

Specifying the locale while connected

The setLocale() method may be used to indicate the locale in which text will be returned to the client for the session, while it is connected.

If the locale has not been set for the session while connecting or the setLocale() method has not been called, then the session locale is determined as follows:

1. The value of the SM_LOCALE environment variable.
2. The default is en_US.

Retrieving and setting log, error and trace levels at runtime

There are three computed attributes available to get and set the log, error, and trace levels of a Domain Manager at runtime. These computed attributes, described in Computed attributes to retrieve and set log, error, and trace levels at runtime, are available on the SM_System object.

Table 3-1. Computed attributes to retrieve and set log, error, and trace levels at runtime

Computed Attributes	Description
logLevel	The minimum exception level for sending messages to the system error logger.The logLevel attributeis a string, and can be any one of the values set for the **--**loglevel command line option:
errLevel	The minimum exception level for writing messages to the log files. The errLevel attributeis a string, and can be any one of the values set for the **--**errlevel command line option.
traceLevel	Used to print a stack trace to the SDK log file when an exception at this level or above occurs. Exceptions below this level do not write a stack trace. The traceLevel attributeis a string, and can be any one of the values set for the **--**tracelevel command line option.

The values of these computed attributes can be retrieved and set, and valid values are:

- None
- Emergency
- Alert
- Critical
- Error
- Warning
- Notice
- Informational,
- Debug

Note Fatal is a synonym for Critical.

Retrieving the current level

You can retrieve the current levels of SM_System::SM-System::logLevel, SM_System::SM-System::errLevel, or SM_System::SM-System::traceLevel. A string is returned which represents the current level, such as "Warning", "Error", or "Fatal". For example:

```
my $smsystem = $session->object("SM_System","SM-System");
my $curr_error_level = $sm_system->{errLevel};
```

Setting the level

To change the current levels, obtain a pointer to the object, and then set the value of SM_System::SM-System::logLevel, SM_System::SM-System::errLevel, or SM_System::SM-System::traceLevel to the appropriate level.

In this example, the trace level setting is changed to None.

```
my $smsystem = $session->object("SM_System","SM-System");
$sm_system->{traceLevel} = "None";
```

When you change the log, error or trace levels a message is printed in the log file. The log message will appear similar to the following:

```
[April 8, 2009 5:03:41 PM EDT +122ms] t@1149000000 SM_ProtocolEngine-6
JM_MSG-*-JM_TRACE_LEVEL_CHANGED-User 'user1', using remote dmctl client (id 6), on host host1 with
credentials tpadmin1 has changed the Trace level to None; in file
"/mypath/repos/jiim/SM_JIIM_Support_Impl.c" at line 458
```

Wrapper functions

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

save

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

The save() function 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 the put() 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 following 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 depends on the usual Perl concept of array or scalar context, as well as the definition of 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 by using the dmctl utility, as shown:

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

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

- Example 1:

```
$obj = $session->object( "Router::gw1" );
$fan= $obj->findFan( 2 );
```

- Example 2:

```
$fan = $session->invoke( "Router::gw1", "findFan", 2 );
```

- Example 3:

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

invoke_t

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

The invoke_t() function is 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.

Note The RegEx version that is supported is the UTF-8 and Unicode regexp compliant engine (coming from ICU).

The default is no flags, therefore, the search uses glob pattern matching 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, for example, SM_System::SM-System), 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 that explain 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> ],
   ...
  ]
  ```

  Example:

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

getClosure

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

The getClosure() function returns a list of 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 that explain 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> ],
   ...
  ]
  ```

  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 that explain $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> ],
   ...
  ]
  ```

getExplainedBy

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

The getExplainedBy() function is the inverse of the getExplains() function.

It returns those problems (or events) which the MODEL developer has listed as explaining this event.

The $onehop parameter is optional:

- If it is omitted or passed as FALSE, the full list of problems that explain $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> ],
 ...
]
```

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 that represent 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 or a more mnemonic string as shown in Subscription flag parameter values

Table 3-2. 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)
- “e” means subscribe to events.

  Note If “p”, “a” or “e” are not present, “p” is assumed.

- “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 a 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 those described in Transaction lock options.

Table 3-3. 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	The behavior of READ_LOCK is 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 that ensures that no other script can see the intermediate stage with an incompletely deleted suite of objects.

delete

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

The delete() function deletes the specified object instance from the repository.

Note This does not clean up all the object interdependencies and links. For a cleaner object deletion, use the remove() operation, if one exists, for the object class in question. The section invoke also provides additional information to the related 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, the getEventType() function returns a string that describes the type of the event. The possible strings returned are described in Event types.

Table 3-4. 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( );
```

The getServerName() function returns the name of the Domain Manager to which the session is connected.

insertElement

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

The insertElement() function 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] );
```

The removeElement() function 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] );
```

Primitives

4

This chapter includes the following topics:

- Primitive naming conventions
- Primitive calling conventions
- Error handling
- Error codes
- Data types
- Primitives

Primitive naming conventions

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 Smarts Console, and the Domain Manager.

Name

InCharge::primitives

Conventions

The names given to the primitives follow a convention of using lowercase, except for the first letter of the second and subsequent words of multiword names. For example, to get operation arguments, the name of the primitive is 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 described in the following sections.

Since primitives are designed to be called by using 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 getInstances() is called getLeaf 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 VMware Smart Assurance software, on the other hand, uses function names that look like sm_property_unsubscribe(). They start with “sm_” and use all lowercase 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 by using InCharge::session->object(), InCharge::session->create(), or InCharge::session->getInstances().

The general approach used for calling primitives is as follows:

1 Initialize a session and obtain a reference to it.

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

2 Call the primitives required, by 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 Domain Manager 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,

4 Establish a session.

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

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

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

6 Manipulate the object by using the reference.

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

7 Close the session.

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

Error handling

All the functions and methods of objects in the API modules throw errors by 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 error message is rendered in the locale set by the client session.

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 Error handling. 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

Error codes provides a description of the different error codes and their associated types.

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, such as “[5][301]”
6	Bad argument	Argument content or type error, or invalid name, invalid option, or wrong type, such as a scalar arguement being passed but a reference was required
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

Table 4-1. Error codes (Continued)

Error code	Error type	Description
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, in the following sections, 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

The $session data type is a reference to a valid InCharge::session object - created by 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 $object data type is the specification of a repository object to be acted upon.

This can be given in one of the formats described in Formats to specify a repository object.

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" );`

Table 4-2. Formats to specify a repository object (Continued)

Format	Description
$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, $instance	This format uses two parameters. The first parameter contains the Perl *undef* value, to indicate that it is unknown. This causes the API to perform a query to determine that name of the object's class. This syntax can only be used to refer to existing objects. `$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

The @objects data type is a list of objects is to be returned, which is only used as a return type.

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

$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 by using one of the following syntaxes:

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

  The first syntax is used where the list is held in an array variable. The second syntax 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 in Symptom data codes. 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, as shown in the following example. 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 one of the built-in TYPE methods of the InCharge::session module, as shown:

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

  Type codes describes the type code values that are used.

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

Table 4-4. Type codes (Continued)

Constant	Literal	Description
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
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 )
```

The classExists function 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 codebook.

correlate

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

The correlate() function triggers the ``Code book'' correlation actions, where symptoms are analyzed and correlated into problems.

countChildren

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

The countChildren() function counts the child classes of the specified class.

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

countClassInstances

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

The countClassInstances() function 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( )
```

The countClasses() function counts the number of classes present in the system.

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

countElements

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

The countElements() function counts the number of elements in the specified relationship.

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

countInstances

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

The countInstances() function counts the total number of objects in the repository, of all classes.

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

countLeafInstances

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

The countLeafInstances() function 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 )
```

The countf() function 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. The section $freshness provides additional information.

createInstance

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

The createInstance() function 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 )
```

Deletes the specified object instance from the repository. Note that this does not clean up all the object interdependencies and links. When the Domain Manager has a MODEL based on ICIM, for a cleaner object deletion, you can use the remove() operation, if one exists for the object class in question. The section invoke provides additional information.

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

deleteObserver

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

The deleteObserver() function is an alias for purgeObserver(). The section purgeObserver provides additional information.

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

The deleteObserver() function reverses the effect of getObserverId(), deregistering the script as an observer.

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

eventIsExported

The eventIsExported() function is an alias for getEventExported(). The section getEventExported provides additional information.

execute

The execute() function is an alias for executeProgram(). The section executeProgram provides additional information.

executeProgram

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

The executeProgram() function is an alias for execute(). The section execute provides additional information.

The executeProgram() function executes an VMware Smart Assurance 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

The exists() function is an alias for instanceExists(). The section instanceExists provides additional information.

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 those classes that match are also returned.

0x100000 = Glob. This causes the match to be done by 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 )
```

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() returns a value, get_T() returns a type code and value in a two-element array.

getAggregationEvents

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

The getAggregationEvents() function 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 )
```

The getAllEventNames() function is an alias for getEvents(). The section getEvents provides additional information.

The getAllEventNames() function 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( )
```

The getAllInstances() function gets the names of all instances present in the ICIM database.

Note The getAllInstances() function can potentially return a very large array and should not be used.

getAllProperties and getAllProperties_t

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

The getAllProperties() function 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 following examples:

- The first example:

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

- The second example:

  ```
  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 in the following example:

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

getArgDirection

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

The getArgDirection() function 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, which is beyond the scope of the API, dmctl, and ASL.

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

getArgType

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

The getArgType() function is an alias for getOpArgType() and getOperationArgumentType(). The sections getOpArgType and getOperationArgumentType provide additional information.

The getArgType() function gets the type of the specified argument for the specified class operation. The section $type, @types describes the possible data types.

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

getAttributes

The getAttributes() function is an alias for getAttributeNames(). The section getAttributeNames provides additional information.

getAttributeNames

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

The getAttributeNames() function is an alias for getAttributes(). The section getAttributes provides additional information.

The getAttributeNames() function 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 )
```

The getAttributeTypes() function 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]. The section $type, @types provides a description of the possible values.

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

getByKey

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

The getByKey() function 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. The section $freshness provides additional information.

```
@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. The section $freshness provides additional information.

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

getChildren

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

The getChildren() function gets the list of classes that are child classes of a specified one, that is, classes derived from the base class.

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

getClassDescription

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

The getClassDescription() function 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( );
```

The getClassHierarchy() function 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, as described in Class hierarchy descriptor.

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 )
```

The getClassInstances() function 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( )
```

The getClasses() function 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( )
```

The getCorrelationParameters() function returns a nine-element array, each element of which contains a parameter relating to the Domain Manager correlation mechanism.

The array elements are described in getCorrelationParameters return values:

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 )
```

The getEnumVals() function returns the list of strings that represent the possible values for an enumerated property.

The returned list of strings 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 a property that is not an enumerated one, an error is thrown.

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

getEvents

The getEvents() function is an alias for getAllEventNames(). The section getAllEventNames provides additional information.

getEventCauses

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

The getEventCauses() function 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, that is, 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 )
```

The getEventExplainedBy() function 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 )
```

The getEventExported() function is an alias for eventIsExported(). The section eventIsExported provides additional information.

Returns one of the following:

- 1 if the specified event is exported by the class
- 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 )
```

The getEventNames() function 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 )
```

The getEventSymptoms() function 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 )
```

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

Table 4-7. getEventType return codes

Return code	Event type
0	Event
1	Aggregation
2	Symptom
3	Causality
4	Problem

Table 4-7. getEventType return codes (Continued)

Return code	Event type
5	Imported event
6	Propagated Aggregation
7	Propagated Symptom
8	Same type

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

getInstances

The getInstances() function is an alias for getClassInstances(). The section getClassInstances provides additional information.

getInstrumentationType

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

The getInstrumentationType() function returns the instrumentation type for a specified object.

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

getLeafInstances

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

The getLeafInstances() function is an alias for getInstances(). The section getInstances provides additional information.

The getLeafInstances() function 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( )
```

The getLibraries() function is an alias for getModels(). The section getModels provides additional information.

The getLibraries() function gets the list of libraries loaded into the system.

getModels

The getModels() function is an alias for getLibraries(). The section getLibraries provides additional information.

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

  For example:

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

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

getObserverId

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

The getObserverId() function creates and returns a new observer ID.

The deleteObserver() primitive reverses this action. The section deleteObserver provides additional information.

getOpArgType

The getOpArgType() function is an alias for getArgType(). The section getArgType provides additional information.

getOpArgs

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

The getOpArgs() function is an alias for getOperationArguments(). The section getOperationArguments provides additional information.

The getOpArgs() function 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 )
```

The getOpDescription() function is an alias for getOperationDescription(). The section getOperationDescription provides additional information.

The getOpDescription() function returns a textual description of the specified class operation.

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

getOperationArguments

The getOperationArguments() function is an alias for getOpArgs(). The section getOpArgs provides additional information.

getOperationArgumentType

The getOperationArgumentType() function is an alias for getArgType(). The section getArgType provides additional information.

getOperationDescription

The getOperationDescription() function is an alias for getOpDescription(). The section getOpDescription provides additional information.

getOperationFlag

The getOperationFlag() function is an alias for getOpFlag(). The section getOpFlag provides additional information.

getOperationReturnType

The getOperationReturnType() function is an alias for getOpReturnType(). The section getOpReturnType provides additional information.

getOperations

The getOperations() function is an alias for getOpNames(). The section getOpNames provides additional information.

getOpFlag

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

The getOpFlag() function is an alias for getOperationFlag(). The section getOperationFlag provides additional information.

The getOpFlag() function gets the flag associated with the specified class operation.

The value returned is between 0 and 3, as defined getOpFlag return codes.

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 )
```

The getOpNames() function is an alias for getOperations(). The section getOperations provides additional information.

The getOpNames() function gets the list of operations for the specified class.

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

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

getOpReturnType

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

The getOpReturnType() function is an alias for getOperationReturnType(). The section getOperationReturnType provides additional information.

The getOpReturnType() function returns the return type code for the specified class operation.

By 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 in getOpReturnType.

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

getParentClass

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

The getParentClass() function 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 )
```

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

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

getProblemExplanation

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

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 )
```

The getProblemNames() function gets the event names of problems associated with the specified class.

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

getProblemSymptomState

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

The getProblemSymptomState() function 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( )
```

The getPrograms() function gets the list of ``programs'' that are running in the Domain Manager.

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

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

getPropAccess

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

The getPropAccess() function returns a number that indicates the level of access to the specified property.

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

The return value effectively identifies the method by which the property value is obtained internally. Possible values and their meanings are listed in getPropAccess return codes .

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

getPropDescription

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

The getPropDescription() function is an alias for getPropertyDescription(). The section getPropertyDescription provides additional information.

The getPropDescription() function returns a textual description of the named class property.

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

getProperties

The getProperties() function is an alias for getPropNames(). The section getPropNames provides additional information.

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

The getPropertyDescription() function is an alias for getPropDescription(). The section getPropDescription provides additional information.

getProperties

The getProperties() function is an alias for getPropNames(). The section getPropNames provides additional information.

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

getPropertyType

The getPropertyType() function is an alias for getPropType(). The section getPropType provides additional information.

getPropIsReadonly

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

The getPropIsReadonly() function 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 )
```

The getPropIsRelationship() function 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 )
```

The getPropIsRequired() function indicates whether the specified class property is required to have a value.

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

getPropNames

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

The getPropNames() function retrieves the names of all the properties of a given class.

getPropRange

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

The getPropRange() function 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 )
```

The getPropType() function is an alias for getPropertyType(). The section getPropertyType

Returns the data type for a specified class property.

The section $type, @types 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 )
```

The getPropertySubscriptionState() function gets the current state of subscription to the specified event.

The possible reply values are listed in getPropertySubscriptionState return codes.

Table 4-10. getPropertySubscriptionState return codes

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

getRelatedClass

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

The getRelatedClass() function 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 )
```

The getRelationNames() function is an alias for getRelations(). The section getRelations provides additional information.

The getRelationNames() function gets the names of all the relationship properties for the specified class.

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

getRelations

The getRelations() function is an alias for getRelationNames(). The section getRelationNames provides additional information.

getRelationTypes

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

The getRelationTypes() function returns a list of type numbers for the relationships which are returned by the getRelationNames() call.

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

The section $type, @types describes the possible data types.

getReverseRelation

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

The getReverseRelation() function 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 )
```

The getSubscriptionState() function gets the current state of subscription to the specified event.

The possible values are listed in getSubscriptionState return codes .

Table 4-11. getSubscriptionState return codes

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

getThreads

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

The getThreads() function returns a list of threads that run 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 as they are described in getThreads return codes:.

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 thread IDs and names of all threads in thread ID order.

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

getf

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

The getf() function gets the contents of the specified property of the object with reference to its freshness. The section $freshness provides additional information.

The return type is scalar, array, or array reference, as appropriate, as described in *"Data types" on page 70*.

```
$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(), the getf_t() function returns the contents of the specified property but *getf_t()* also returns a code for the type of the data. The section $freshness provides additional information.

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);
```

The getfAllProperties() function is the same as getAllProperties(), but takes the freshness of the values into account, and refreshes any stale properties before returning the results, that is, those that are older than $freshness seconds. The section $freshness provides additional information.

The sectiongetAllProperties and getAllProperties_t provides a description of the $flag.

getfMultipleProperties and getfMultipleProperties_t

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

The getfMultipleProperties() function is like getMultipleProperties(), but refreshes values that are staler than $freshness seconds and need re-polling. The section $freshness provides additional information.

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 ])
```

The InsertElement_P() function inserts something into a relationshipset.

In order to access the low-level primitive version of this call, you must invoke it by 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 )
```

The InstanceExists() function is an alias for exists(). The section exists provides additional information.

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

The invoke() function is an alias for invokeOperation(). The section invokeOperation provides additional information.

invoke_t and invoke_T

The invoke_t() and invoke_T() functions are an alias for invokeOperation_t(). The section invokeOperation_t and invokeOperation_T.

invokeOperation

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

The invokeOperation() function is an alias for invoke(). The section invoke provides additional information.

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

The invokeOperation() function 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. For example:

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

invokeOperation_t and invokeOperation_T

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

The invokeOperation_t() function is an alias for invoke_t(). The section invoke_t and invoke_T provide additional information.

The invokeOperation_t() function 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 )
```

The isAbstract() function 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 )
```

The isBaseOf() function returns TRUE if $class2 is a base class of $class1, that is, $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 )
```

The isBaseOfOrProxy() function 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 )
```

The isInstrumented() function indicates whether the specified class has associated instrumentation.

```
$flag = $session->isInstrumented( "TCPConnect" );
```

isMember

```
$boolean = $session->isMember( $object1, $relation, $object2)
```

The isMember() function 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] )
```

The isMemberByKey() function 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)
```

The isMemberByKeyf() function is similar to isMemberByKey(), but with reference to the freshness of the value. The section $freshness provides additional information.

```
$exists = $session->isMemeberByKeyf(
  "GA_CompoundDriver::Bridge-Generic-Driver",
  "drivers", [ "INT", 10 ], 120 )
```

isMemberf

```
$boolean = $session->isMemberf( $object1, $relation,
     $object2, $freshness )
```

The isMemberf() function 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. The section $freshness provides additional information.

```
$flag = $session->isMemberf( "Router::strrtbos",
    "ComposedOf",
    "Interface::IF-strrtbos/1",
    240 );
```

isSubscribed

```
$boolean = $session->isSubscribed( $object, $event )
```

The isSubscribed() function returns TRUE if the specified event has been subscribed to by the calling process.

```
$subscribed = $session->isSubscribed( "Router::gw1", "Down");
```

loadLibrary

```
$session->loadLibrary( $library )
```

The loadLibrary() function is an alias for loadModel(). The section loadModel provides additional information.

The loadLibrary() function loads a library, model, into sm_server memory.

```
 $session->loadLibrary( $libname );
```

loadModel

The loadModel() function is an alias for loadLibrary(). The section loadLibrary provides additional information.

loadProgram

```
$session->loadProgram( $program )
```

The loadProgram() function loads the named program into sm_server memory.

```
 $session->loadProgram( "dmdebug" );
```

noop

```
$session->noop()
```

The noop() function is an alias for ping(). The section ping provides additional information.

This is a type of ping. It sends a null command string to the Domain Manager, and thus determines whether the client/server link is active.

notify

The section notify provides a description of notify().

ping

The ping() function is an alias for noop(). The section noop provides additional information.

propertySubscribe

```
$session->propertySubscribe( $object, $property, $interval )
```

The propertySubscribe() function subscribes to notifications of changes to the specified object property. *"Event subscription" on page 22* provide an overview of subscribing to events in a Domain Manager.

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);
```

The propertySubscribeAll() function 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(). The section propertyUnsubscribeAll provides additional information.

```
$session->propertySubscribeAll( 0, "Router", "gw1",
     ".*", 30 );
```

propertyUnsubscribe

```
$session->propertyUnsubscribe( $object, $property )
```

The propertyUnsubscribe() function reverses the effect of the propertySubscribe() call. The section propertySubscribe provides additional information.

```
$session->propertyUnsubscribe( "Router::gw1", "Vendor" );
```

propertyUnsubscribeAll

```
$session->propertyUnsubscribeAll( $flags, $class_pattern,
      $instance_pattern,
      $property_pattern );
```

The propertyUnsubscribeAll() function 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

The purgeObserver() function is an alias for deleteObserver(). The section deleteObserver provides additional information.

put_P

```
$session->put_P( $object, $property, [ $type, $value ] )
```

The put_P() function writes the specified value to the specified object property.

The put_P() function 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 examples are essentially equivalent:

- The first example

  ```
  $obj = $session->object( "Router::gw2" );
  $obj->{Vendor} = "Cisco";
  ```

- The second example

  ```
  $obj->put( "Vendor", "Cisco" );
  ```

- The third example

  ```
  $ojb->put( Vendor => "Cisco", PrimaryOwnerContact => "Joe Blog" );
  ```

- The fourth example

  ```
  $session->put( "Router::gw", "Vendor", "Cisco" );
  ```

- The fifth example

```
$session->put_P( "Router::gw", "Vendor", [ "STRING", "Cisco" ] );
```

quit

```
$session->quit( )
```

The quit() function is an alias for shutdown(). The section shutdown provides additional information.

The quit() function closes down the Domain Manager cleanly, saving the configured parts of the repository to disk.

removeElement_P

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

The removeElement_P() function removes an element from an object relationship, such as ComposedOf.

In order to access the low-level primitive version of this call, invoke it by 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])
```

The removeElementByKey() function removes a set-valued property by key.

restoreRepository

```
$session->restoreRepository( $filename, $purgeflag )
```

The restoreRepository() function restores the repository from file, optionally purging existing repository contents in the process.

```
$session->restoreRepository( "save.rps", 0 );
```

setCorrelationParameters

```
$session->setCorrelationParameters( @info )
```

The setCorrelationParameters() function sets the Domain Manager correlation parameters.

The section getCorrelationParameters describes the fields of the @info array. The following example sets the correlation interval to 20 seconds.

```
@info = $session->getCorrelationParameters( );
$info[1] = 20;
$session->setCorrelationParameters( @info );
```

shutdown

The shutdown() function is an alias for quit(). The section quit provides additional information.

storeAllRepository

```
$session->storeAllRepository( $filename )
```

The storeAllRepository() function 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 )
```

The storeClassRepository() function saves the repository for the named class in the specified file.

```
$session->saveClassRepository( "save.rps", "Host" );
```

subscribeEvent

```
$session->subscribeEvent( $object, $event )
```

The subscribeEvent() function subscribes to a specific event without using wildcard pattern matching, unlike subscribeAll().

The function unsubscribeEvent() cancels subscriptions that were established by using subscribeEvent(). The section unsubscribeEvent provides additional information.

```
$session->subscribeEvent( "Router::gateway39", "Down" );
```

subscribeAll

Note Use InCharge::session->subscribe() instead.

topologySubscribe

```
$session->topologySubscribe( )
```

The topologySubscribe() function subscribes to notifications of topology updates.

The subscription/observer mechanism is described in detail in *"Event subscription" on page 22*. API subscriptions topology subscriptions may be reversed by using topologyUnsubscribe().

topologyUnsubscribe

```
$session->topologyUnsubscribe( )
```

The topologyUnsubscribe() function cancels topology subscriptions previously requested with the topologySubscribe() function. The section topologySubscribe provides additional information.

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

transactionAbort

```
$session->transactionAbort( )
```

Use InCharge::session>abortTxn instead.

The transactionAbort() function aborts a transactional block previously started by using transactionStart(). The section transactionStart provides additional information.

transactionCommit

```
$session->transactionCommit( )
```

The transactionCommit() function commits a transactional block previously started using transactionStart(). The section transactionStart provides additional information

transactionStart

```
$session->transactionStart( $lock_code )
```

The transactionStart() function starts a transaction block, which may subsequently be aborted or committed by using transactionAbort()/transactionCommit().

```
 sub SM_READ_LOCK { 1 };
```

$session->transactionStart(SM_READ_LOCK);

The $lock_code values have the possible values shown in Lock code literals.

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

unsubscribeAll

Note Use InCharge::session->unsubscribe() instead.

unsubscribeEvent

```
$session->unsubscribeEvent( $object, $event )
```

The unsubscribeEvent() function unsubscribes from the event previously subscribed by using subscribeEvent(). The section subscribeEvent provides additional information.

IPv6 Considerations

5

This chapter includes the following topics:

- Conventions for specifying IPv6 addresses
- Controlling name resolution

Conventions for specifying IPv6 addresses

Internet Protocol version 6 (IPv6) uses colons (:) in its addresses instead of periods (.), which are used in Internet Protocol version 4 (IPv4) .

Sometimes when you use VMware Smart Assurance APIs or command-line utilities, you need to specify an IP address with a port number. The port number is delimited by a colon (:). (The combination of an IP address and port number is also called a *socket*.)

For the IPv6 address and port number to be interpreted correctly, specify the IPv6 address by using *one* of the following conventions:

- Enclose the IPv6 address within a pair of double quotation marks and square brackets. The syntax is:

  ```
  "[ipv6_address]:port"
  "[2001:0db8::0010]:65000"
  ```

- Enclose the IPv6 address with a pair of back slashes and square brackets. The syntax is:

  ```
  \[ipv6_address\]:port
  \[2001:0db8::0010\]:65000
  ```

- If the port is a default port, omit the port number and specify only the IPv6 address. No additional convention notations are needed.

  For example, for an IPv6 address and the default port of 162, specify:

  ```
  2001:0db8::0010
  ```

Controlling name resolution

The order in which name resolution is performed depends on how you specify a hostname and whether the SM_IP_VERSIONS environment variable is set.

When a user specifies a hostname for an VMware Smart Assurance utility or the Perl API, the behavior occurs in the following order:

1 If the hostname includes an explicit Internet Protocol (IP) protocol (the suffix to the right of the colon), the hostname is resolved to an address of that protocol. For example:
   - frame.someDomain.vmware.com:v4 —Resolves to an IPv4 address.
     - frame:v6 —Resolves to an IPv6 address.
     - frame:v4v6 —Resolves to an IPv4 address, or, if that fails, to an IPv6 address.
     - frame:v6v4 —Resolves to an IPv6 address, or, if that fails, to an IPv4 address.
2 If the hostname does not include an explicit IP protocol, the utility searches for the SM_IP_VERSIONS environment variable and uses the setting specified for variable. The SM_IP_VERSIONS environment variable is described in The SM_IP_VERSIONS environment variable.
3 If the environment variable is not set and the IP protocol is not explicitly provided, the default behavior is to resolve the hostname as an IPv6 address, or, if that fails, to an IPv4 address (the behavior for the v6v4 suffix).

   Additional information about discovery and name resolution is provided in the VMware Smart Assurance IP Management Suite Discovery Guide and the VMware Smart Assurance IP Manager Deployment Guide.

The SM_IP_VERSIONS environment variable

The SM_IP_VERSIONS environment variable enables you to control the Internet Protocol (IP) version used for name resolution.

This affects VMware Smart Assurance utilities that use a command line (for example, dmctl), some ASL scripts, the Perl API, and DNS lookup of undiscovered hostnames.

The variable can be set depending on the order in which you want to do name resolution. If the variable is not set, and the IP protocol is not explicitly provided (for example, frame.someDomain.vmware.com:v4), the default behavior is to resolve the hostname as an IPv6 address, or, if that fails, to an IPv4 address. The variable should be set to the Internet Protocol version that is predominate for the network.

To set this variable, add it to the runcmd_env.sh file, which is located in the BASEDIR/smarts/local/conf directory of the product suite.

The syntax of the environment variable is:

```
SM_IP_VERSIONS="ip_value"
```

Acceptable values for the SM_IP_VERSIONS environment variable lists acceptable values for the SM_IP_VERSIONS environment variable.

Table 5-1. Acceptable values for the SM_IP_VERSIONS environment variable

ip_value	Description
"V4"	Hostname is resolved to an IPv4 address.
"V6"	Hostname is resolved to an IPv6 address.
"V4V6"	Hostname is resolved to an IPv4 address. If that fails, the Domain Name System server tries to resolve the hostname to an IPv6 address.
"V6V4"	Hostname is resolved to an IPv6 address. If that fails, the Domain Name System server tries to resolve the hostname to an IPv4 address (default).
Note: The acceptable value can also be lowercase ("v4", "v6", "v4v6" or "v6v4").	

Detailed instructions about setting environment variables and information about the runcmd_env.sh file is provided in the VMware Smart Assurance System Administration Guide.