InCharge™

Perl Reference Guide

Version 1.0

Cisco Part Number: OL-6126-01

the technology of Crystal Decisions. The use of all Crystal Decisions software products is subject to a separate license agreement included with the Software electronically, in written materials, or both. YOU MAY NOT USE THE CRYSTAL DECISIONS SOFTWARE UNLESS AND UNTIL YOU READ, ACKNOWLEDGE AND ACCEPT THE TERMS AND CONDITIONS OF THE CRYSTAL DECISIONS' SOFTWARE LICENSE AGREEMENT. IF YOU DO NOT ACCEPT THE TERMS AND CONDITIONS OF THE CRYSTAL DECISIONS' SOFTWARE LICENSE, YOU MAY RETURN, WITHIN THIRTY (30) DAYS OF PURCHASE, THE MEDIA PACKAGE AND ALL ACCOMPANYING ITEMS (INCLUDING WRITTEN MATERIALS AND BINDERS OR OTHER CONTAINERS) RELATED TO THE CRYSTAL DECISIONS' TECHNOLOGY, TO SMARTS FOR A FULL REFUND; OR YOU MAY WRITE, CRYSTAL WARRANTIES, P.O. BOX 67427, SCOTTS VALLEY, CA 95067, U.S.A.

GNU eTeks PJA Toolkit
Copyright © 2000-2001 Emmanuel PUYBARET/eTeks info@eteks.com. All Rights Reserved.
The eTeks PJA Toolkit is resident on the CD on which the Software was delivered to you. Additional information is available at eTEks' web site: http://www.eteks.com. The eTeks PJA Toolkit program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License (GPL) as published by the Free Software Foundation; version 2 of the License. The full text of the applicable GNU GPL is available for viewing at http://www.gnu.org/copyleft/gpl.txt. You may also request a copy of the GPL from the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA. The eTeks PJA Toolkit program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

For a period of three years from the date of your license for the Software, you are entitled to receive under the terms of Sections 1 and 2 of the GPL, for a charge no more than SMARTS' cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code for the GNU eTeks PJA Toolkit provided to you hereunder by requesting such code from SMARTS in writing: Attn: Customer Support, SMARTS, 44 South Broadway, White Plains, New York 10601.

IBM Runtime for AIX
The Software contains the IBM Runtime Environment for AIX(R), Java™ 2 Technology Edition Runtime Modules © Copyright IBM Corporation 1999, 2000 All Rights Reserved.

HP-UX Runtime Environment for the Java™ 2 Platform
The Software contains the HP-UX Runtime for the Java™ 2 Platform, distributed pursuant to and governed by Hewlett-Packard Co. ("HP") software license terms set forth in detail at: http://www.hp.com. Please check the Software to determine the version of Java runtime distributed to you.

DataDirect Technologies
Portions of this software are copyrighted by DataDirect Technologies, 1991-2002.

NetBSD
Copyright © 2001 Christopher G. Demetriou. All rights reserved. Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. All advertising materials mentioning features or use of this software must display the following acknowledgement:
   This product includes software developed for the NetBSD Project. See http://www.netbsd.org/ for information about NetBSD.
4. The name of the author may not be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. <<Id: LICENSE, v 1.2 2000/06/14 15:57:33 cgd Exp>>

RSA Data Security, Inc.
Copyright © 1991-2, RSA Data Security, Inc. Created 1991. All rights reserved. License to copy and use this software is granted provided that it is identified as the "RSA Data Security, Inc. MD5 Message-Digest Algorithm" in all material mentioning or referencing this software or this function. License is also granted to make and use derivative works provided that such works are identified as "derived from the RSA Data Security, Inc. MD5 Message-Digest Algorithm" in all material mentioning or referencing the derived work. RSA Data Security, Inc. makes no representations concerning either the merchantability of this software or the suitability of this software for any particular purpose. It is provided "as is" without express or implied warranty of any kind. These notices must be retained in any copies of any part of this documentation and/or software.

AES
Copyright © 2003, Dr Brian Gladman <brg@gladman.me.uk>, Worcester, UK. All rights reserved.

License Terms:

The free distribution and use of this software in both source and binary form is allowed (with or without changes) provided that:

1. distributions of this source code include the above copyright notice, this list of conditions and the following disclaimer;
2. distributions in binary form include the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other associated materials;
3. the copyright holder's name is not used to endorse products built using this software without specific written permission.

ALTERNATIVELY, provided that this notice is retained in full, this product may be distributed under the terms of the GNU General Public License (GPL), in which case the provisions of the GPL apply INSTEAD OF those given above.

Disclaimer: This software is provided 'as is' with no explicit or implied warranties in respect of its properties, including, but not limited to, correctness and/or fitness for purpose. Issue Date: 26/08/2003

Contents

Preface **xiii**

Intended Audience xiii
Prerequisites xiii
Document Organization xiii
Documentation Conventions xiv
Additional Resources xv
InCharge Commands xv
Documentation xv
Common Abbreviations and Acronyms xvi
Technical Support xvii
xviii

1 Introduction **1**

Fundamental Concepts 1
Classes **2**
Instances **2**
Properties - Attributes and Relationships **2**
Operations **3**
Events **3**
Primitives - Basic InCharge Manager Interface **3**
Overview of a Simple API Perl Script 4
Using Primitives and Object References 5
Event Subscription 6
Registering an Observer 7
Subscribing to Notifications. 8
Receiving Notifications 9

2 Object 17

Name 17
Synopsis 17
Description 17
Functions and Methods 20
object 20
get 20
get_t 21
put 22
isNull 23
invoke 23
invoke_t 23
insertElement 23
removeElement 24
delete 24
notify 24
clear 24
countElements 24

3 Session 27

Name 27
Synopsis 27
Description 28
Function Groups 28
Session Management 28
InCharge Primitives 28
Utility Functions 28
Compatibility Functions 29
Error Handling 29
Session Management Functions 29
new 29
init 31
broken 32
reattach 32

detach 33
observer 33
receiveEvent 34
object 34
create 35
callPrimitive 36
Utility Functions 37
TYPE Function 37
getFileno 37
getProtocolVersion 37
primitiveIsAvailable 37
Compatibility Functions 38
save 38
put 38
invoke 39
invoke_t 40
findInstances 40
getCauses 41
getClosure 41
getExplains 42
getExplainedBy 43
subscribe and unsubscribe 43
transaction, abortTxn and commitTxn 45
delete 46
getEventType 46
getServerName 47
insertElement 47
removeElement 47

4 Primitives **49**
Name 49
Description 49
Primitive Naming Conventions 49
Primitive Calling Conventions 50

Error Handling 52
Error codes 54
Data Types 55
$session 55
$object 55
@objects 56
$symptom, @symptoms 56
$symptomData, @symptomData 57
$type, @types 57
$freshness 59
Primitives 59
classExists 59
consistencyUpdate 59
correlate 59
countChildren 60
countClassInstances 60
countClasses 60
countElements 60
countInstances 60
countLeafInstances 60
countf 61
createInstance 61
deleteInstance 61
deleteObserver 61
eventIsExported 62
execute 62
executeProgram 62
exists 62
findInstances_P 62
forceNotify 63
get 63
get_t and get_T 64
getAggregationEvents 64
getAllEventNames 64
getAllInstances 65

getAllProperties and getAllProperties_t 65
getArgDirection 65
getArgType 66
getAttributes 66
getAttributeNames 66
getAttributeTypes 67
getByKey 67
getByKey_t and getByKey_T 67
getByKeyf 67
getByKeyf_t and getByKeyf_T 68
getChildren 68
getClassDescription 68
getClassHierarchy 68
getClassInstances 69
getClasses 69
getCorrelationParameters 69
getEnumVals 70
getEvents 70
getEventCauses 70
getEventClassName 71
getEventDescription 71
getEventExplainedBy 71
getEventExported 71
getEventNames 72
getEventSymptoms 72
getEventType_P 72
getInstances 73
getInstrumentationType 73
getLeafInstances 73
getLibraries 73
getModels 74
getMultipleProperties and getMultipleProperties_t 74
getObserverId 74
getOpArgType 74

getOpArgs 74
getOpDescription 75
getOperationArguments 75
getOperationArgumentType 75
getOperationDescription 75
getOperationFlag 75
getOperationReturnType 75
getOperations 75
getOpFlag 76
getOpNames 76
getOpReturnType 76
getParentClass 77
getProblemClosure 77
getProblemExplanation 77
getProblemNames 77
getProblemSymptomState 77
getPrograms 78
getPropAccess 78
getPropDescription 78
getProperties 79
getPropertyDescription 79
getProperties 79
getPropertyType 79
getPropIsReadonly 79
getPropIsRelationship 80
getPropIsRequired 80
getPropNames 80
getPropRange 80
getPropType 80
getPropertySubscriptionState 81
getRelatedClass 81
getRelationNames 81
getRelations 81
getRelationTypes 82

getReverseRelation 82
getSubscriptionState 82
getThreads 82
getf 83
getf_t and getf_T 83
getfAllProperties and getfAllProperties_t 84
getfMultipleProperties and getfMultipleProperties_t 84
hasRequiredProps 84
insertElement_P 84
instanceExists 85
invoke 85
invoke_t and invoke_T 85
invokeOperation 85
invokeOperation_t and invokeOperation_T 86
isAbstract 86
isBaseOf 87
isBaseOfOrProxy 87
isInstrumented 87
isMember 87
isMemberByKey 87
isMemberByKeyf 88
isMemberf 88
isSubscribed 88
loadLibrary 88
loadModel 88
loadProgram 89
noop 89
notify 89
ping 89
propertySubscribe 89
propertySubscribeAll 89
propertyUnsubscribe 90
propertyUnsubscribeAll 90
purgeObserver 90

put_P 90
quit 91
removeElement_P 91
removeElementByKey 91
restoreRepository 91
setCorrelationParameters 92
shutdown 92
storeAllRepository 92
storeClassRepository 92
subscribeEvent 92
subscribeAll 93
topologySubscribe 93
topologyUnsubscribe 93
transactionAbort 93
transactionCommit 93
transactionStart 94
unsubscribeAll 94
unsubscribeEvent 94

Index 95

Preface

Intended Audience

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

Prerequisites

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

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

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

Document Organization

This guide consists of the following chapters.

1. INTRODUCTION	Gives an overview of the InCharge features accessible to a programmatic interface and a discussion of how the Perl programming language may be used to interact with an InCharge Manager.
2. OBJECT	Provides a description of the API facilities available that allow objects in the InCharge repository to be manipulated in an object-oriented style, similar to the ASL language.
3. SESSION	Provides a description of the API facilities available to create Perl scripts to establish client/server connections and to obtain object references, which can be used to manipulate the objects in the topology.
4. PRIMITIVES	Provides a description of the Perl low-level remote calls supported by the API. These primitives provide the standard protocol between client application

Table 1: Document Organization

Documentation Conventions

Several conventions may be used in this document as shown in Table 2.

CONVENTION	EXPLANATION
`sample code`	Indicates code fragments and examples in Courier font
keyword	Indicates commands, keywords, literals, and operators in bold
`%`	Indicates C shell prompt
#	Indicates C shell superuser prompt
<parameter>	Indicates a user-supplied value or a list of non-terminal items in angle brackets
[option]	Indicates optional terms in brackets

CONVENTION	EXPLANATION
/InCharge	Indicates directory path names in italics
yourDomain	Indicates a user-specific or user-supplied value in bold, italics
File > Open	Indicates a menu path in italics
▲▼	Indicates a command that is formatted so that it wraps over one or more lines. The command must be typed as one line.

Table 2: Documentation Conventions

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

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

Finally, unless otherwise specified, the term InCharge Manager is used to refer to InCharge programs such as Domain Managers, Global Managers, and adapters.

Additional Resources

In addition to this manual, SMARTS provides the following resources.

InCharge Commands

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

Documentation

Readers of this manual may find other SMARTS documentation (also available in the ***BASEDIR****/smarts/doc/pdf* directory) helpful.

InCharge Documentation

The following SMARTS documents are product independent and thus relevant to users of all InCharge products:

- *InCharge Release Notes*
- *InCharge Documentation Roadmap*
- *InCharge System Administration Guide*
- *InCharge Operator's Guide*
- *InCharge ASL Reference Guide*
- *InCharge Perl Reference Guide*

Software Development Kit Documentation

The following SMARTS documents are relevant to users of the Software Development Kit.

- *InCharge Software Development Kit Remote API for Java* (in html format)
- *InCharge Software Development Kit Remote API Programmer's Guide* (in pdf format)
- *InCharge Software Development Kit ICIM Reference* (in html format)
- *InCharge Software Development Kit MODEL Reference Guide* (in pdf format)

Common Abbreviations and Acronyms

The following lists common abbreviations and acronyms that are used in the InCharge guides.

ASL	Adapter Scripting Language
CDP	Cisco Discovery Protocol
ICIM	InCharge Common Information Model
ICMP	Internet Control Message Protocol
IP	Internet Protocol
MSFC	Multilayer Switch Feature Card
MIB	Management Information Base
MODEL	Managed Object Definition Language

RSFC	Router Switch Feature Card
RSM	Router Switch Module
SNMP	Simple Network Management Protocol
TCP	Transmission Control Protocol
VLAN	Virtual Local Area Network

Technical Support

SMARTS provides technical support by e-mail or phone during normal business hours (8:00 A.M.—6:00 P.M. U.S. Eastern and Greenwich Mean Time). In addition, SMARTS offers the InCharge Express self-service web tool. The web tool allows customers to access a personalized web page and view, modify, or create help/trouble/support tickets. To access the self-service web tool, point your browser to:

https://websupport.smarts.com/SelfService/smarts/en-us

U.S.A Technical Support

E-Mail: *support@smarts.com*
Phone: +1.914.798.8600

EMEA Technical Support

E-Mail: *support-emea@smarts.com*
Phone: +44 (0) 1753.878140

Asia-Pac Technical Support

E-Mail: *support-asiapac@smarts.com*

You may also contact SMARTS at:

	U.S.A World Headquarters	United Kingdom
Address	SMARTS 44 South Broadway White Plains, New York 10601 U.S.A	SMARTS Gainsborough House 17-23 High Street Slough Berkshire SL1 1DY United Kingdom
Phone	+1.914.948.6200	+44 (0)1753.878110
Fax	+1.914.948.6270	+44 (0)1753.878111

For sales inquiries, contact SMARTS Sales at:
sales@smarts.com.

SMARTS is on the World Wide Web at:
http://www.smarts.com

1

Introduction

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

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

Note: This API is compatible with InCharge Manager Versions 6.0 and later.

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

Fundamental Concepts

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

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

Classes

Classes describe the objects that are modeled for use in an InCharge Manager. For example, Router is a class, and all routers that are managed by an InCharge 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 which are similar in nature but not in detail.

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

Instances

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

Properties - Attributes and Relationships

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

1 Attributes

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

2 Relationships

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

Operations

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

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

Events

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

Primitives - Basic InCharge Manager Interface

Primitives are Perl functions that provide the basic interface between a client application and the InCharge 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. Example of primitives that are not normally used directly are *get()* and *invoke()*. These are the InCharge calls that allow an instance's properties to be queried and its operations to be called. In both the API for Perl and ASL, these calls are normally invoked by using a classic object-oriented syntax.

For more information about the InCharge data structures, refer to the *InCharge Software Development Kit ASL Reference Guide*, chapter 7.

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

Overview of a Simple API Perl Script

The general approach to writing a script using the API for InCharge is to follow these basic steps.

1 Open a session

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

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

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

2 Work with the domain

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

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

3 Close the session

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

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

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

1 Establish an InCharge session

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

2 Obtain an object reference

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

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

3 Manipulate the object

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

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

4 Close the InCharge 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 InCharge Managers. Each of these primitives can be invoked with reference to the *InCharge::session* handle (see above), and takes arguments that exactly match the InCharge API's syntax.

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

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

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

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

```
    print $router . " - " . $vendor . "\n";
}
```

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

Event Subscription

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

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

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

The second mechanism provides asynchronous notifications via subscriptions and is used when the client program needs to listen for events generated by the InCharge 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 InCharge servers to send subscribed events to a client program, the client must first register itself with the InCharge Manager as an event observer.

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

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

Here are some example code fragments that achieve this goal.

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

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

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

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

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

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

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

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

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

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

Subscribing to Notifications.

Once the client has registered itself as an observer, the next step is to inform the InCharge Manager about which events it is interested in receiving. InCharge allows clients to subscribe to a number of different types of events. These are listed in the following table.

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

Table 3: Subscription Methods Summary

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

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

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

Receiving Notifications

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

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

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

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

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

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

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

If no time-out is specified, the call waits forever.

The first element of the @event array, accessed using the Perl syntax: $event[0], contains the event's time stamp measured using normal unix time_t semantics; i.e., the number of seconds since midnight January 1st, 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 InCharge Manager when the status of an event changes. The format of the notification record is:

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)

Table 4: Notification Record - NOTIFY

Normally, the InCharge 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 InCharge 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 InCharge Manager. The InCharge Manager sends NOTIFY messages in both cases. The API keeps internal notes about active events, and changes the event type accordingly.

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)

Table 5: Notification Record - CERTAINTY_CHANGE

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

EVENT RECORD ENTRY	DESCRIPTION
$event[0]	Timestamp (INTEGER)
$event[1]	"CLEAR"
$event[2]	Object class name (STRING)
$event[3]	Object instance name (STRING)
$event[4]	Event name (STRING)

Table 6: Notification Record - CLEAR

Object Create/Delete Records

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

EVENT RECORD ENTRY	DESCRIPTION
$event[0]	Timestamp (INTEGER)
$event[1]	"CREATE"
$event[2]	Object class name (STRING)
$event[3]	Object instance name (STRING)

Table 7: Notification Record - CREATE

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

EVENT RECORD ENTRY	DESCRIPTION
$event[0]	Timestamp (INTEGER)
$event[1]	"DELETE"
$event[2]	Object class name (STRING)
$event[3]	Object instance name (STRING)

Table 8: Notification Record - DELETE

Class Load Records

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

EVENT RECORD ENTRY	DESCRIPTION
$event[0]	Timestamp (INTEGER)
$event[1]	"CLASS_LOAD"
$event[2]	Class name (STRING)

Table 9: Notification Record - CLASS_LOAD

Relation/Property Change Records

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

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)

Table 10: Notification Record - RELATION_CHANGE

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

EVENT RECORD ENTRY	DESCRIPTION
$event[0]	Timestamp (INTEGER)

EVENT RECORD ENTRY	DESCRIPTION
$event[1]	"ATTR_CHANGE"
$event[2]	Object class name (STRING)
$event[3]	Object instance name (STRING)
$event[4]	Property name (STRING)

Table 11: Notification Record - ATTR_CHANGE

InCharge Manager Connect/Disconnect Records

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

EVENT RECORD ENTRY	DESCRIPTION
$event[0]	Timestamp (INTEGER)
$event[1]	"DISCONNECT"
$event[2]	Domain name (STRING)

Table 12: Notification Record - DISCONNECT

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

Subscription status records

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

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)

Table 13: Notification Record - ACCEPT, REJECT, (PROPERTY)

Event suspension records

Under certain circumstances, the InCharge 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:

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)

Table 14: Notification Record - SUSPEND (PROPERTY)

Time-out records

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

EVENT RECORD ENTRY	DESCRIPTION
$event[0]	Timestamp (INTEGER)
$event[1]	"TIMEOUT"
$event[2]	Domain name (STRING)

Table 15: Notification Record - TIMEOUT

2

Object

Name

InCharge::object - Access to SMARTS InCharge repository objects.

Synopsis

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

Description

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

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

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

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

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

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

The option of omitting the class name does not work with the *InCharge::session->create()* method because InCharge 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 that the API does additional work to determine the object's class, if you choose not to provide the class name in these calls, imposing a slight performance penalty.

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

```
$vendor = $obj->get( "Vendor" );
$vendor = $obj->{Vendor};
($vendor,$model) = $obj->get( "Vendor", "Model" );
%properties = $obj->get( );
$obj->put( "Vendor", "Cisco" );
$obj->{Vendor} = "Cisco";
$obj->put(Vendor => "Cisco", Model => "2010" );
```

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

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

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

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

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

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

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

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

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

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

Instead, use:

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

or,

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

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

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

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

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

Functions and Methods

object

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

creates a new object reference.

get

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

or,

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

or,

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

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

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

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

Table 16: Return Type for Call Syntax and Perl Evaluation Context

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

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

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

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

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

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

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

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

get_t

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

or,

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

or,

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

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

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

The third syntax gets the types and values for all the properties and relations of the object and stores them in a hash, indexed by the property names. This approach has a significant performance benefit when working with InCharge Manager versions 6 and above.

Example:

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

This example will print:

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

put

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

This method allows fields of the object to be modified in the InCharge 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 to InCharge's native ASL language.

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

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

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

or

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

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

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

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

isNull

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

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

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

invoke

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

Invokes the named repository class operation on the object.

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

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

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

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

invoke_t

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

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

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

insertElement

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

Inserts the specified objects into an object relationship.

One or more can be specified to be inserted.

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

removeElement

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

Removes the specified items from an object relationship.

One or more items can be specified to be removed.

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

delete

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

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

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

notify

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

Notifies the specified event for the object.

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

clear

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

Clear the specified event for the object.

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

countElements

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

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

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

3

Session

Name

InCharge::session - SMARTS InCharge Manager session management.

Synopsis

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

(... and continuing with the methods, described below ...)

Description

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

Refer to the introductory chapter for an overview of this and the other InCharge::* modules and a simple tutorial description of how they are used.

Function Groups

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

Session Management

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

InCharge Primitives

The *InCharge::session* module permits access to the low-level primitive functions of the InCharge 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 InCharge Perl scripts.

Compatibility Functions

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

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

Error Handling

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

Session Management Functions

The following session management functions are provided.

new

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

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

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

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

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

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

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

The default domain name is INCHARGE.

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

4 user => $user_name

5 username => $user_name

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

There is no default username.

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

6 password => $password

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

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

7 description => $description

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

8 traceServer => 1

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

9 timeout => $timeout

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

If only the domain name is given, it can be specified without the domain=> key.

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

init

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

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

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

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

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

If it encounters a command line syntax error, it calls usageError, in the main script, which the developer must provide. A single large text string containing a description of the standard options handled is passed as the argument to usageError, allowing the author to include information about the standard options as well as any non standard 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 saving and restoring the contents @ARGV first, as in the example:

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

broken

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

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

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

reattach

$session->reattach( );

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

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

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

This function should be called after a [13] I/O Error is thrown by any of the InCharge Manager access calls in order to shutdown and reopen the socket, cleaning up the protocol. If this step is not taken there is a danger that residual packets on the connection would cause synchronization problems between the client and InCharge Manager. See the Primitives/ErrorHandling discussion for an explanation of the [13] prefix.

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

detach

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

Detaches from the InCharge domain referred to by *$session*.

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

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

This call does not completely destroy the *$session* reference contents but retains enough information to allow the session to be re-established. Thus, it is possible to call *$session->*`reattach()` to re-connect to the InCharge Manager using the same parameters as were used in the initial connection. However, the event subscriptions need to be re-established explicitly in this event.

observer

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

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

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

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

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

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

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

receiveEvent

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

Listen for subscribed events from the InCharge Manager.

The received events are returned as an array or, in scalar context, a reference to an array containing three or more elements. The details of the significance of the different events are described in the subscriptions section.

The first element of all events is the time stamp, on the InCharge 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"; e.g., 0.25 = a quarter second.

object

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

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

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

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

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

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

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

1. object('Router::gw1')

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

2 object('Router', 'gw1')

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

3 object('::gw1')

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

4 object(undef, 'gw1')

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

5 object('gw1')

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

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

create

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

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

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

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

callPrimitive

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

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

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

The following are equivalent, although the first is preferred.

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

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

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

Utility Functions

TYPE Function

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

Converts an InCharge Manager data type mnemonic string to its internal numeric code, or vise versa. So the following prints "13".

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

The following code prints "STRING".

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

getFileno

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

Returns the underlying system file number the refers to the socket used for the script/server connection.

This is useful when the script wishes to use the Perl *select* statement to listen for events from multiple domain servers, using multiple observer objects.

getProtocolVersion

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

Returns the protocol version number supported by the InCharge Manager.

This is a single integer number derived by the following calculation.

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

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

primitiveIsAvailable

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

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

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

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

Compatibility Functions

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

save

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

Saves the repository in the specified file.

If a class name is specified then only the instances of that class are saved.

put

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

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

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

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

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

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

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

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

invoke

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

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

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

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

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

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

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

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

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

invoke_t

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

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

The value is a Perl scalar, if the operation returns a scalar, or an array reference, if the operation returns an array. The type will contain one of the InCharge 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 InCharge glob pattern matching.

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

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

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

Example:

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

getCauses

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

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

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

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

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

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

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

Example:

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

getClosure

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

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

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

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

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

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

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

Example:

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

getExplains

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

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

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

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

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

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

getExplainedBy

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

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

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

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

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

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

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

subscribe and unsubscribe

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

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

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

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

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

Table 17: Subscription Flag Parameter Values

As a compatibility aid, the $flag can also be specified as a string of letters. In this case, each of the letters are subscription qualifiers: 'p' means subscribe to problems, “a” means subscribe to aggregates (impacts), and “e” means subscribe to events. If none of these are present, “p” is assumed. The letter “v” means run in verbose mode, which turns on subscription control messages. The action of these options is the same as that provided by the sm_adapter program’s –subscribe= option.

Examples:

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

transaction, abortTxn and commitTxn

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

Transactions, Commit and Abort.

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

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

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

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

or

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

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

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

A keyword can be any one of the following:

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

Table 18: Transaction Lock Options

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

The API aborts any open transactions when the script terminates.

Example:

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

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

delete

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

Deletes the specified object instance from the repository.

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

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

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

or

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

getEventType

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

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

EVENT TYPE LITERAL	DESCRIPTION
EVENT	Event
AGGREGATION	Aggregation
SYMPTOM	Symptom
PROBLEM	Problem
UNKNOWN	Error indication

Table 19: Event Types

Example:

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

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

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

getServerName

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

Returns the name of the InCharge Manager to which the InCharge session is connected.

insertElement

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

Inserts one or more elements into an object relationship.

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

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

removeElement

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

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

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

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

4

Primitives

Name

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

Description

This section lists and describes the InCharge Manager primitives, the 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 InCharge Console and the InCharge Manager. This manual page describes the Perl syntax for calling these primitives and provides outline documentation on their use and semantics.

Primitive Naming Conventions

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

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

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

Where a primitive returns a value that may be of any type, a second version of the call is provided that returns both the numeric type code and the return value. The name of this extended version is the same as the lesser original but with "_t" appended. You can also specify "_T" instead of "_t", in which case when the primitive returns an ANYVAL_ARRAY_SET (i.e., a structure of structures), the fields of the structures are also accompanied by their types. This is a reference to a 2-element array containing type and value is given 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 *getLeafInstances()* 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 InCharge, on the other hand, uses function names that look like *sm_property_unsubscribe()*. They start with "sm_" and use all lower case words delimited by underscores. This set of functions is less complete than the C++ equivalent interface and does not provide a one-to-one match of all the InCharge Manager primitives. The API for Perl does not provide a match for the C interface function names.

Primitive Calling Conventions

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

The general approach used is as follows.

1. Initialize an InCharge session and obtain a reference to it.

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

2. Call the primitives required using the session reference; e.g.,

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

1. Establish an InCharge session.

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

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

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

3. Manipulate the object using the reference.

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

4. Close the InCharge session.

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

Error Handling

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

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

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

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

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

Refer to the section on the *eval* and *die* functions in the Perlfunc manual page for more details about using this mechanism.

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 the table below. 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.

An example of a script is provided that attempts a connection with an InCharge 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 CODE	ERROR TYPE	DESCRIPTION
1	Syntax error	Wrong number of arguments, missing argument, or too many arguments
2	System error	System call error; e.g., socket creation failed
3	Connection error	Socket connection error
4	Authentication error	Authentication error
5	HTTP error	Other session init failure (HTTP error in second number; e.g., "[5][301]"
6	Bad argument	Argument content or type error, or invalid name, invalid option, or wrong type; e.g., is scalar but reference wanted
7	Broker error	Cannot attach to broker
8	No domain	Domain not registered with broker
9	Protocol error	Protocol error, data size error, or unsupported protocol format
10	Isolated	Not attached
11	Invalid operation	Invalid or illegal operation
12	Bad function	Bad function call or primitive name
13	IO error	Socket IO error
14	Timeout	Timer expired
15	DM error	Error returned by InCharge 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

Table 20: Error Codes

Data Types

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

$session

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

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

$object

The specification of an repository object to be acted upon.

This can be given in one of the following formats:

'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 a single string, as above, but with the empty class name. In this case the API performs a database query to determine the name of the class to use. This syntax can only be used to refer to existing objects.

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

$class, $instance

This format uses two parameters, where the first contains the class and the second contains the name of the instance.

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

undef, $instance

This format uses two parameters, as above, but with the first replaced by the Perl *undef* value, indicating that it is unknown. This causes the API to perform a query to determine that name of the object's class. This syntax can only be used to refer to existing objects.

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

InCharge::object reference

This format is used whenever an object name is required. It is also possible to pass an *InCharge::object* reference.

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

@objects

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

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

$symptom, @symptoms

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

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

You can gain access to the elements using the syntax,

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

or,

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

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

$symptomData, @symptomData

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

SYMPTOM DATA CODE	DESCRIPTION AND TYPE
$x[0]	state (INTEGER) 0 = active 1 = inactive 2 = suspended 3 = not monitored
$x[1]	last occurance (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)

Table 21: Symptom Data Codes

$type, @types

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

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

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

or,

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

The type code values used are:

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

CONSTANT	LITERAL	DESCRIPTION
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

Table 22: Type Codes

$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 re-calculating or re-polling if the property was last updated more than the specified $freshness seconds ago.

Primitives

classExists

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

Returns 1 if the specified class exists or 0, otherwise.

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

    }
```

consistencyUpdate

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

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

correlate

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

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

countChildren

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

Counts the child classes of the specified class.

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

countClassInstances

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

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

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

countClasses

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

Counts the number of classes present in the system.

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

countElements

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

Counts the number of elements in the specified relationship.

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

countInstances

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

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

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

countLeafInstances

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

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

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

countf

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

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

createInstance

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

Note: Use the *InCharge::session->create()* function instead, as described in the Session chapter of this guide.

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

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

deleteInstance

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

Note: Use the *InCharge::session->delete()* or *InCharge::object->delete()* function instead, as described in the Session or Object chapters of this guide.

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

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

deleteObserver

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

Alias for *purgeObserver()*.

Note: Consider this an internal call - use *$session->`detach()`* instead, as discussed in the Session chapter of this guide.

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

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

eventIsExported

Alias for *getEventExported()*.

execute

Alias for *executeProgram()*.

executeProgram

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

Alias for *execute()*

Executes an InCharge program, passing arguments to it.

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

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

exists

Alias for *instanceExists()*.

findInstances_P

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

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

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

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

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

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

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

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

forceNotify

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

Notifies, or clears, the specified event.

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

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

get

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

Note: Use the features of the *InCharge::object* module instead, as discussed in the Object and Session chapters of this guide.

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

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

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

getAggregationEvents

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

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

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

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

getAllEventNames

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

Alias for *getEvents()*.

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

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

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

getAllInstances

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

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

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

getAllProperties and getAllProperties_t

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

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

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

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

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

or,

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

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

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

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

getArgDirection

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

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

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

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

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

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

getArgType

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

Alias for *getOpArgType()* and g*etOperationArgumentType()*.

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

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

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

getAttributes

Alias for *getAttributeNames()*.

getAttributeNames

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

Alias for *getAttributes()*.

Gets the list of all attributes for the specified class.

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

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

getAttributeTypes

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

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

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

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

getByKey

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

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

Tables are properties that can contain arrays of values.

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

getByKey_t and getByKey_T

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

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

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

getByKeyf

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

Identical to *getByKey()* but takes the "freshness" of the entry into account.

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

getByKeyf_t and getByKeyf_T

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

Identical to *getByKey_t()* but takes the "freshness" of the entry into account.

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

getChildren

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

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

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

getClassDescription

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

Gets a textual description of the class.

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

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

getClassHierarchy

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

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

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

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

Table 23: Class Hierarchy Descriptor

getClassInstances

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

Gets the list of instances of a specified class.

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

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

getClasses

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

Gets the list of class present in the system.

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

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

   }
}
```

getCorrelationParameters

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

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

The array elements have the following significance:

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)

ELEMENT	DESCRIPTION
info[5]	spurious symptom probability (FLOAT)
info[6]	time limit (INT)
info[7]	suspend correlation (BOOLEAN)
info[8]	provide explanation (BOOLEAN)

Table 24: getCorrelationParameters Return Values

getEnumVals

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

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

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

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

getEvents

Alias for *getAllEventNames()*

getEventCauses

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

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

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

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

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

getEventClassName

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

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

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

getEventDescription

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

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

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

getEventExplainedBy

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

Note: Use *InCharge::session*->getExplainedBy instead, as discussed in the Session chapter of this guide.

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

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

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

getEventExported

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

Alias for *eventIsExported()*.

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

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

getEventNames

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

Gets the list of events handled by the specified class.

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

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

getEventSymptoms

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

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

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

getEventType_P

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

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

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

RETURN CODE	EVENT TYPE
0	Event
1	Aggregation
2	Symptom
3	Causality
4	Problem
5	Imported event
6	Propagated Aggregation
7	Propagated Symptom
8	Same type

Table 25: getEventType Return Codes

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

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

getInstances

Alias for *getClassInstances()*.

getInstrumentationType

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

Returns the instrumentation type for a specified object.

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

getLeafInstances

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

Alias for *getInstances()*.

Gets the list of instances of a specified class.

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

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

getLibraries

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

Alias for *getModels()*.

Gets the list of libraries loaded into the system.

getModels

Alias for *getLibraries()*

getMultipleProperties and getMultipleProperties_t

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

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

The syntax of the primitive itself is,

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

or,

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

For example,

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

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

getObserverId

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

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

Creates and returns a new observer id.

The *deleteObserver()* primitive reverses this action.

getOpArgType

Alias for *getArgType()*.

getOpArgs

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

Alias for *getOperationArguments()*.

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

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

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

getOpDescription

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

Alias for *getOperationDescription()*.

Returns a textual description of the specified class operation.

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

getOperationArguments

Alias for *getOpArgs()*.

getOperationArgumentType

Alias for *getArgType()*.

getOperationDescription

Alias for *getOpDescription()*.

getOperationFlag

Alias for *getOpFlag()*.

getOperationReturnType

Alias for *getOpReturnType()*.

getOperations

Alias for *getOpNames()*.

getOpFlag

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

Alias for *getOperationFlag()*.

Gets the flag associated with the specified class operation.

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

RETURN CODE	OPERATION
0	No flag
1	Idempotent
2	Constant
3	Readonly

Table 26: getOpFlag Return Codes

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

getOpNames

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

Alias for *getOperations()*.

Gets the list of operations for the specified class.

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

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

getOpReturnType

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

Alias for *getOperationReturnType()*.

Returns the return type code for the specified class operation.

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

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

getParentClass

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

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

This is the logical inverse of *getChildren()*.

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

getProblemClosure

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

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

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

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

getProblemExplanation

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

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

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

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

getProblemNames

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

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

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

getProblemSymptomState

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

```
                                                    $eventname )
```

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

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

getPrograms

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

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

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

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

getPropAccess

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

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

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

RETURN CODE	PROPERTY ACCESS LEVEL
0	No access
1	Stored
2	Computed
3	Instrumented
4	Propagated
5	Uncomputable
6	Computed with expression

Table 27: getPropAccess Return Codes

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

getPropDescription

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

Alias for *getPropertyDescription()*.

Returns a textual description of the named class property.

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

getProperties

Alias for *getPropNames()*.

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

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

getPropertyDescription

Alias for *getPropDescription()*.

getProperties

Alias for *getPropNames()*.

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

getPropertyType

Alias for *getPropType()*.

getPropIsReadonly

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

Indicates whether or not the specified class property is read-only.

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

getPropIsRelationship

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

Indicates whether or not the specified class property is a relationship.

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

getPropIsRequired

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

Indicates whether or not the specified class property is required to have a value.

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

getPropNames

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

getPropRange

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

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

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

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

getPropType

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

Alias for *getPropertyType()*.

Returns the data type for a specified class property.

See the $type section of the discussion about data types for a description of the possible values. This call always returns the integer number representation of the type.

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

getPropertySubscriptionState

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

Gets the current state of subscription to the specified event.

The possible reply values are:

RETURN CODE	SUBSCRIPTION STATE
0	Unsubscribed
1	Pending
2	Subscribed
3	Suspended

Table 28: getPropertySubscriptionState Return Codes

getRelatedClass

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

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

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

getRelationNames

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

Alias for *getRelations()*.

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

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

getRelations

Alias for *getRelationNames()*.

getRelationTypes

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

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

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

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

getReverseRelation

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

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

The inverse of ComposedOf is PartOf.

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

getSubscriptionState

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

Gets the current state of subscription to the specified event.

The possible values are:

RETURN CODE	SUBSCRIPTION STATE
0	Unsubscribed
1	Pending
2	Subscribed
3	Suspended

Table 29: getSubscriptionState Return Codes

getThreads

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

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

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

RETURN ARRAY ELEMENT	THREAD INFORMATION
$t[0]	Process ID
$t[1]	name
$t[2]	State
$t[3]	Status

Table 30: getThreads Return Codes

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

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

getf

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

Gets the contents of the specified property of the object with reference to its freshness; see the discussion of data types. The return type is scalar, array, or array reference, as appropriate.

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

getf_t and getf_T

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

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

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

getfAllProperties and getfAllProperties_t

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

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

See *getAllProperties()* for a description of $flag.

getfMultipleProperties and getfMultipleProperties_t

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

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

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

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

hasRequiredProps

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

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

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

insertElement_P

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

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

Inserts something into a relationshipset.

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

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

instanceExists

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

Alias for *exists()*.

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

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

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

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

invoke

Alias for *invokeOperation()*.

invoke_t and invoke_T

Alias for *invokeOperation_t()*.

invokeOperation

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

Alias for *invoke()*.

Note: Use the features of the *InCharge::object* module instead, as described in the Object chapter of this guide.

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

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

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

Note: The argument types, such as INT and STRING in the above example, can be specified using either their type names, as shown above, or numeric codes. See $type in the DATA TYPES section for the mapping.

invokeOperation_t and invokeOperation_T

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

Alias for *invoke_t()*.

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

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

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

isAbstract

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

Indicates whether the specified class is abstract or not.

An abstract class is one from which other classes are derived but which cannot have any objects.

```
$class = "ICIM_UnitaryComputerSystem";
$flag = $session->isAbstract( $class );
```

isBaseOf

```
$boolean = $session->isBaseOf( $class1, $class2 )
```

Returns TRUE if $class2 is a base class of $class1; i.e., $class1 is derived from $class2.

Note: For the purposes of this query, all classes are taken to be derived from themselves.

```
$class1 = "Router";
$class2 = "ICIM_UnitaryComputerSystem";
$is_it = $session->isBaseOf( $class1, $class2 );
```

isBaseOfOrProxy

```
$boolean = $session->isBaseOfOrProxy( $class1, $class2 )
```

Returns TRUE (1) if $class2 is a base class or proxy class of $class1.

```
$class1 = "Router";
$class2 = "ICIM_UnitaryComputerSystem";
$is_it = $session->isBaseOfOrProxy( $class1, $class2 );
```

isInstrumented

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

Indicates whether the specified class has associated instrumentation.

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

isMember

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

Returns TRUE if *$object2* is a member of the specified *$object1* relationship.

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

isMemberByKey

```
$boolean = $session->isMemberByKey( $object, $table,
                                    $keytype, $keyvalue] )
```

Indicates whether or not an entry in the named object table exists.

```
$exists = $session->isMemeberByKey(
```

```
                    "GA_CompoundDriver::Bridge-Generic-Driver",
                    "drivers", [ "INT", 10 ] )
```

isMemberByKeyf

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

Like isMemeberByKey, but with reference to the freshness of the value.

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

isMemberf

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

Returns TRUE if *$object2* is a member of the specified *$object1* relationship. If the $relation is a computed or polled value and is more than $freshness seconds old, it is refreshed first.

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

isSubscribed

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

Returns TRUE if the specified event has been subscribed to by the calling process.

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

loadLibrary

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

Alias for *loadModel()*.

Loads a library, model, into sm_server memory.

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

loadModel

Alias for *loadLibrary()*

loadProgram

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

Loads the named program into sm_server memory.

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

noop

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

Alias for *ping()*

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

notify

Information to follow.

ping

Alias for *noop()*

propertySubscribe

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

Subscribes to notifications of changes to the specified object property.

See the introductory discussion of the subscription. The actions of this call are reversed by *propertyUnsubscribe()*.

```
$session->propertySubscribe( "Router::gw1",
                             "Vendor", 30 );
```

propertySubscribeAll

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

Subscribes to changes in all the matching properties in the matching objects.

The meaning of the $flags is described for the *subscribe()* session function.

The actions of this call are reversed by *propertyUnsubscribeAll()*.

```
$session->propertySubscribeAll( 0, "Router", "gw1",
```

```
                                                    ".*", 30 );
```

propertyUnsubscribe

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

Reverses the effect of the *propertySubscribe()* call.

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

propertyUnsubscribeAll

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

Unsubscribes from changes in all the matching properties in the matching objects.

The meaning of the $flags is described for the *subscribe()* session function.

purgeObserver

Alias for *deleteObserver()*.

put_P

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

Writes the specified value to the specified object property.

This is the low-level primitive that the *put()* function of *InCharge::session* uses and is called when using the hash dereferencing syntax of *InCharge::object*.

The reader is encouraged to use the *InCharge::object* logic.

The following are essentially equivalent.

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

or,

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

or,

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

or,

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

or,

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

quit

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

Alias for *shutdown()*.

Closes down the InCharge Manager cleanly, saving the configured parts of the repository to disk.

removeElement_P

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

Note: Use *InCharge::object::removeElement()* instead of this primitive. The "_P" in the name is there to save ambiguity between this primitive and the *InCharge::session* and *InCharge::object* equivalents.

This call removes an element from an object relationship, such as ComposedOf.

In order to access the low-level primitive version of this call, invoke it using the primitive method because the *InCharge::session* module has a method of the same name that provides an enhanced interface.

```
$session->removeElement_P("Router::gw", "ComposedOf",
                      [ "OBJREG", "Host::pingu6" ] );
```

removeElementByKey

```
$session->removeElementByKey( $object, $table,
                          [ $keytype, $keyvalue ] )
```

Remove a set-valued property by key.

restoreRepository

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

Restores the repository from file, optionally purging existing repository contents in the process.

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

setCorrelationParameters

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

Sets the InCharge Manager correlation parameters.

For a description of the fields of the @info array, refer to the *getCorrelationParameters()* primitive. The example below sets the correlation interval to 20 seconds.

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

shutdown

Alias for *quit()*.

storeAllRepository

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

Note: Use the *InCharge::session->save()* method instead.

Saves the repository in the named file, which is located in the directory $SM_HOME/repos. The directory name must not contain any path separator characters.

```
$session->saveAllRepository( "save.rps" );
```

storeClassRepository

```
$session->storeClassRepository( $filename, $class )
```

Note: Use the *InCharge::session->save()* method instead.

Save the repository for the named class in the specified file.

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

subscribeEvent

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

Subscribes to a specific event without using wildcard pattern matching, unlike *subscribeAll()*.

The function *unsubscribeEvent()* cancels subscriptions established using *subscribeEvent()*.

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

subscribeAll

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

topologySubscribe

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

Subscribes to notifications of topology updates.

The subscription/observer mechanism is described in detail in the introductory discussion of subscriptions. API subscriptions topology subscriptions may be reversed using *topologyUnsubscribe()*.

topologyUnsubscribe

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

Cancels topology subscriptions previously requested with the *topologySubscribe()* function.

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

transactionAbort

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

Use *InCharge::session*>abortTxn instead.

Aborts a transactional block previously started using *transactionStart()*.

transactionCommit

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

Note: Use *InCharge::session->commitTxn()* instead.

Commits a transactional block previously started using *transactionStart()*.

transactionStart

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

Note: Use *InCharge::session->transaction()* instead.

Starts a transaction block, which may subsequently be aborted or committed using *transactionAbort()/transactionCommit()*. The $lock_code values have the following possible values:

LOCK CODE	LITERAL
0	SM_NO_LOCK
1	SM_READ_LOCK_ONLY
2	SM_READ_LOCK
3	SM_WRITE_LOCK

Table 31: Lock Code Literals

```
sub SM_READ_LOCK { 1 };
$session->transactionStart( SM_READ_LOCK );
```

unsubscribeAll

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

unsubscribeEvent

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

Note: Use *InCharge:session::unsubscribe()* instead.

Unsubscribes from the event previously subscribed using subscribeEvent().

Index

Symbols

$freshness 59
$object 55
$session 55
$symptom 56
$type 57
@objects 56
@symptoms 56
@types 57

A

abortTxn 45, 93
API xiv, 1, 2, 3, 4, 5, 10, 18, 30, 35, 46, 49, 50, 52, 55, 56, 66, 79, 93
ASL xiv, 1, 3, 5, 9, 17, 19, 22, 28, 29, 33, 35, 38, 41, 42, 43, 49, 50, 66, 73, 85
Attribute 1, 2, 65, 66, 67

B

broken 32

C

callPrimitive 36, 38, 39
Certainty 10, 57
Class 1, 2, 3, 10, 11, 12, 13, 14, 18, 19, 23, 35, 36, 38, 39, 40, 41, 42, 43, 46, 55, 56, 57, 58, 59, 60, 61, 62, 63, 66, 68, 69, 71, 72, 73, 75, 76, 77, 78, 79, 80, 81, 84, 85, 86, 87, 92
classExists 59
clear 24
commitTxn 45, 46, 93
consistencyUpdate 59
correlate 59, 60
countChildren 60
countClasses 60
countClassInstances 60
countElements 24, 25, 55, 56, 60, 61
countf 61
countInstances 60
countLeafInstances 60, 61
create 18, 27, 35, 36, 61
createInstance 36, 61

D

delete 24, 46, 61
deleteInstance 61
deleteObserver 61, 62, 74, 90
detach 4, 5, 7, 33, 51, 62
die 13, 29, 52, 53
dmctl 1, 19, 20, 28, 29, 39, 49, 50, 63, 66, 79

E

eval 29, 52
eventIsExported 62, 71
execute 62
executeProgram 62
exists 62, 85

F

findInstances 40, 41, 62
findInstances_P 62, 63
forceNotify 63

G

get 3, 17, 18, 19, 20, 21, 51, 52, 63, 64, 65, 74, 79
get_T 64
get_t 21, 22, 64, 65, 74
getAggregationEvents 64
getAllEventNames 64, 65, 70, 72
getAllInstances 65
getAllProperties 65, 84
getAllProperties_t 65
getArgDirection 65, 66
getArgType 66, 74, 75
getAttributeNames 66, 67
getAttributes 66
getAttributeTypes 66, 67
getByKey 67
getByKey_t 67, 68
getByKeyf 67
getByKeyf_T 68
getCauses 41
getChildren 3, 68, 77
getClassDescription 68
getClasses 4, 28, 51, 69

getClassInstances 8, 55, 60, 69, 73
getClosure 41, 42, 77
getCorrelationParameters 69, 70, 92
getEnumVals 70
getEventCauses 70, 77
getEventClassName 71
getEventDescription 71
getEventExplainedBy 71
getEventExported 62, 71, 72
getEventNames 65, 72
getEvents 64, 70
getEventSymptoms 72
getEventType 46, 47, 72, 73
getEventType_P 72
getExplainedBy 3, 43, 71
getExplains 42, 43, 77
getf 83
getf_T 83
getf_t 83
getfAllProperties 84
getfAllProperties_t 84
getFileno 37
getfMultipleProperties 84
getfMultipleProperties_t 84
getInstances 3, 5, 6, 28, 36, 50, 51, 73
getInstrumentationType 73
getLeafInstances 50, 61, 69, 73
getLibraries 73, 74
getMultipleProperties 38, 74, 79, 84
getMultipleProperties_t 74
getObserverId 62, 74
getOpArgs 50, 74, 75
getOpArgType 66, 74
getOpDescription 75
getOperationArguments 49, 50, 75
getOperationArgumentType 66, 75
getOperationDescription 75
getOperationFlag 75, 76
getOperationReturnType 75, 76
getOperations 75
getOpFlag 75, 76
getOpNames 75, 76
getOpReturnType 75, 76
getParentClass 77
getProblemClosure 70, 77
getProblemExplanation 77
getProblemNames 77
getProblemSymptomState 77
getPrograms 78
getPropAccess 78
getPropDescription 78, 79
getProperties 79
getPropertyDescription 78, 79
getPropertySubscriptionState 81
getPropertyType 79, 80
getPropIsReadonly 79
getPropIsRelationship 80
getPropIsRequired 80
getPropNames 79, 80
getPropRange 80
getPropType 79, 80, 81
getProtocolVersion 37
getRelatedClass 81
getRelationNames 81, 82
getRelations 81
getRelationTypes 82
getReverseRelation 82
getServerName 47
getSubscriptionState 8, 82
getThreads 82, 83

H

hash 5, 6, 19, 20, 21, 22, 65, 90
hasRequiredProps 84

I

ICIM 8, 19, 61, 63, 65, 68
InCharge
  Client xiv, 1, 6, 7, 8, 13, 14, 28, 30, 31, 32, 33, 34, 49, 89
  Manager xiv, 1, 2, 3, 5, 9, 10, 11, 12, 13, 14, 27, 28, 29, 30, 31, 33, 34, 35, 37, 41, 42, 45, 47, 49, 50, 54, 57, 59, 66, 69, 78, 92
  Server xiv, 1, 6, 7, 8, 13, 28, 29, 30, 31, 32, 33, 37, 82, 89
init 4, 6, 7, 8, 27, 31, 32, 46, 51, 52, 55
insertElement 23, 24, 47, 84
insertElement_P 84, 85
Instance 1, 2, 3, 4, 10, 11, 12, 13, 14, 18, 19, 35, 36, 38, 40, 41, 42, 43, 46, 55, 56, 57, 58, 59, 61, 62, 65, 69, 73, 85
instanceExists 62, 85
invoke 3, 17, 19, 23, 39, 40, 46, 57, 61, 85
invoke_T 85
invoke_t 23, 40, 85, 86
invokeOperation 39, 51, 85, 86
invokeOperation_T 86
invokeOperation_t 85, 86
isAbstract 86
isBaseOf 87

isBaseOfOrProxy 87
isInstrumented 87
isMember 87
isMemberByKey 87
isMemberByKeyf 88
isMemberf 88
isNull 23, 85
IsSubscribed 8
isSubscribed 88

L

loadLibrary 88
loadModel 88
loadProgram 89

M

Model 1, 2, 6, 12, 42, 43, 71, 77, 88

N

new 4, 7, 19, 27, 29, 31, 33, 52, 55
noop 89
Notification 7, 8, 9, 10, 11, 12, 13, 14, 15, 43, 89, 93
notify 24, 89

O

object 6, 17, 18, 20, 22, 23, 27, 34, 35, 36, 38, 39, 46, 51, 52, 56, 64, 90
Observer 7, 8, 9, 32, 33, 34, 37, 62, 74, 93
observer 6, 7, 8, 33, 34, 74
openObjectId papi_Introduction.fm
2 454639 v
2 469494 v
2 469498 v
2 469501 v
2 469503 v
2 469512 v
2 469526 v
2 469889 v
2 470472 v
2 470487 v
2 470517 v
2 471036 v
2 471125 v
2 472125 v
openObjectId papi_IX.fm
2 10489 xii
openObjectId papi_Object.fm
2 454639 v
2 469490 v
2 469500 vi
2 469511 vi
2 469563 vi
2 469564 vi
2 469567 vi
2 469605 vi
2 469622 vi
2 469625 vi
2 469635 vi
2 469642 vi
2 469648 vi
2 469651 vi
2 469655 vi
2 469659 vi
2 470095 vi
2 470948 vi
openObjectId papi_PREFACE.fm
2 435167 v
2 436900 v
2 438216 v
2 438222 v
2 675034 v
2 675365 v
2 675367 v
2 675369 v
2 675425 v
2 675464 v
openObjectId papi_Primitives.fm
2 454639 vii
2 469490 vii
2 469500 vii
2 469503 vii
2 469511 vii
2 469536 vii
2 469596 viii

2 469600 viii
2 469618 viii
2 469620 viii
2 469631 viii
2 469646 viii
2 469684 viii
2 469688 viii
2 469694 viii
2 469697 viii
2 469700 viii
2 469706 viii
2 469710 viii
2 469714 viii
2 469718 viii
2 469722 viii
2 469726 viii
2 469729 viii
2 469734 viii
2 469739 viii
2 469745 viii
2 469747 viii
2 469749 viii
2 469757 viii
2 469759 viii
2 469775 viii
2 469782 viii
2 469792 viii
2 469802 viii
2 469808 viii
2 469813 viii
2 469816 viii
2 469829 ix
2 469837 ix
2 469844 ix
2 469846 ix
2 469851 ix
2 469855 ix
2 469863 ix
2 469871 ix
2 469880 ix
2 469889 ix
2 469894 ix
2 469898 ix
2 469904 ix
2 469908 ix
2 469916 ix
2 469928 ix
2 469932 ix
2 469934 ix
2 469942 ix
2 469946 ix
2 469950 ix
2 469957 ix
2 469964 ix
2 469968 ix
2 469972 ix
2 469987 ix
2 469989 ix
2 469995 ix
2 470000 ix
2 470004 ix
2 470006 ix
2 470017 ix
2 470021 ix
2 470023 ix
2 470028 x
2 470035 x
2 470037 x
2 470039 x
2 470041 x
2 470043 x
2 470045 x
2 470047 x
2 470057 x
2 470062 x
2 470069 x
2 470073 x
2 470080 x

2 470086 x
2 470090 x
2 470094 x
2 470098 x
2 470109 x
2 470114 x
2 470118 x
2 470120 x
2 470123 x
2 470125 x
2 470133 x
2 470139 x
2 470143 x
2 470145 x
2 470151 x
2 470156 x
2 470165 x
2 470169 x
2 470174 x
2 470176 x
2 470181 x
2 470187 xi
2 470194 xi
2 470205 xi
2 470210 xi
2 470215 xi
2 470219 xi
2 470225 xi
2 470229 xi
2 470239 xi
2 470245 xi
2 470247 xi
2 470249 xi
2 470263 xi
2 470277 xi
2 470282 xi
2 470288 xi
2 470294 xi
2 470298 xi
2 470306 xi
2 470314 xi
2 470323 xi
2 470333 xi
2 470337 xi
2 470342 xi
2 470344 xi
2 470348 xi
2 470352 xi
2 470354 xi
2 470356 xi
2 470360 xi
2 470370 xi
2 470374 xi
2 470381 xi
2 470383 xi
2 470403 xii
2 470407 xii
2 470417 xii
2 470421 xii
2 470425 xii
2 470431 xii
2 470433 xii
2 470438 xii
2 470443 xii
2 470447 xii
2 470449 xii
2 470452 xii
2 470456 xii
2 470460 xii
2 470464 xii
2 470474 xii
2 470476 xii
2 476403 vii
2 476478 viii
2 478542 viii
openObjectId papi_Session.fm
2 454639 vi
2 469490 vi

2 469500 vi
2 469515 vi
2 469519 vi
2 469521 vi
2 469523 vi
2 469526 vi
2 469528 vi
2 469537 vi
2 469539 vi
2 469565 vi
2 469582 vi
2 469585 vi
2 469591 vi
2 469596 vi
2 469604 vii
2 469609 vii
2 469629 vii
2 469634 vii
2 469645 vii
2 469646 vii
2 469657 vii
2 469662 vii
2 469673 vii
2 469675 vii
2 469678 vii
2 469701 vii
2 469722 vii
2 469735 vii
2 469750 vii
2 469763 vii
2 469774 vii
2 469785 vii
2 469815 vii
2 469853 vii
2 469861 vii
2 469873 vii
2 469876 vii
2 469881 vii
2 472952 vi
2 472972 vii
2 473478 vii
Operation 1, 2, 3, 4, 5, 18, 19, 23, 24, 35, 39, 40, 46, 49, 50, 51, 54, 61, 66, 75, 76, 85

P

Perl xiii, xiv, 1, 3, 4, 5, 6, 7, 9, 10, 13, 20, 21, 22, 28, 29, 30, 35, 37, 40, 46, 49, 50, 52, 56, 65
Perlfunc 52
ping 89
Primitive xiv, 3, 4, 5, 28, 29, 30, 31, 36, 37, 38, 39, 46, 47, 49, 50, 51, 54, 55, 57, 59, 70, 72, 73, 74, 77, 79, 84, 85, 90, 91, 92
primitiveIsAvailable 37
Property 2, 3, 4, 5, 8, 12, 13, 14, 18, 19, 20, 21, 22, 35, 38, 39, 50, 51, 59, 63, 64, 65, 66, 67, 70, 74, 78, 79, 80, 81, 82, 83, 84, 89, 90, 91
propertySubscribe 6, 8, 19, 89, 90
propertySubscribeAll 89
propertyUnsubscribe 8, 89, 90
propertyUnsubscribeAll 89, 90
purgeObserver 61, 90
put 17, 18, 19, 22, 36, 38, 39, 51, 90, 91
put_P 38, 90, 91

Q

quit 91, 92

R

reattach 32, 33
receiveEvent 6, 9, 14, 33, 34
Relationship 1, 2, 12, 22, 23, 24, 38, 41, 42, 46, 47, 60, 61, 80, 81, 82, 87, 88, 91
relationship 2, 87
Relationshipset 2, 84
removeElement 23, 24, 47, 48, 91
removeElement_P 91
removeElementByKey 91
restoreRepository 91, 92

S

save 38, 92
setCorrelationParameters 92
shutdown 91, 92
SM_AUTHORITY 30, 31
SM_HOME 31, 92
storeAllRepository 28, 92
storeClassRepository 92

Subscribe 8, 33, 43, 44
subscribe 8, 19, 43, 44, 89, 90, 93
subscribeAll 8, 93
subscribeEvent 92, 93, 94
Subscription xiii, 6, 7, 8, 13, 32, 33, 34, 44, 81, 82, 89, 93

T

Technical Support xvii
Timestamp 10, 11, 12, 13, 14
topologySubscribe 8, 93
topologyUnsubscribe 8, 93
transaction 45, 46, 94
transactionAbort 93, 94
transactionCommit 93, 94
transactionStart 93, 94

U

Unsubscribe 8, 43
unsubscribe 8, 43, 44, 94
unsubscribeAll 8, 94
unsubscribeEvent 93, 94