Network Manager IP Edition
Version 3 Release 9

Perl API Guide

SC27-2769-01

Network Manager IP Edition
Version 3 Release 9

Perl API Guide

SC27-2769-01

Note

Before using this information and the product it supports, read the information in"Notices" on page 179.

Edition notice

This edition applies to version 3 release 9 of IBM Tivoli Network Manager IP Edition (product number 5724-S45) and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

About this publication v
Intended audience v
What this publication contains v
Publications vi
Accessibility ix
Tivoli technical training. x
Support information x
Conventions used in this publication x

Chapter 1. Overview of the Perl API . . . 1
RIV module overview 1
RIV::Agent module overview 3
RIV::App module overview 7
RIV::OQL module overview 7
RIV::Param module overview 8
RIV::Record module overview 10
RIV::RecordCache module overview 11
RIV::SnmpAccess module overview 12
NCP modules overview 13
NCP::DBI_Factory module overview 14
NCP::Domain module overview 15
Synchronization with message broker. 16
Installing the Perl API 16
Perl builds. 17
Obtaining SNMP information from a network device 17
Perl API modules reference page syntax 18

Chapter 2. Writing discovery agents . . 19
Before you write a discovery agent 19
Writing a discovery agent 20
Example discovery agents 24
Discovery agent skeleton 24
Network entity discovery agent example . . . 26
IP routing discovery agent example 26
Prototype agent definition file template 31
Using threads in discovery agents 33
Discovery agent threads example 33
Default number of threads 34

Chapter 3. Accessing component databases 35
Object Query Language 35
Differences between OQL and Structured Query Language 35
Actions that can be performed on component databases 36
Example Perl scripts that operate on component databases 37
The oql_example.pl example script 37
OQL example script 39

Chapter 4. Performing SNMP queries 41
Using get methods to obtain SNMP information from a device. 41
Making synchronous and asynchronous SNMP get requests 42
Example SNMP GET access script 42
Declare Perl API modules and variables 42
Create and initialize a RIV::Param object. . . . 43
Create and initialize a RIV::App object 44
Create and initialize RIV::SnmpAccess object . . 44
Check the device IP address and node name . . 45
Determine which SNMP GET requests to run . . 46
Perform asynchronous SNMP GET requests . . 46
Perform synchronous SNMP GET requests . . . 47
Print the SNMP varops 48

Chapter 5. Writing and integrating Perl applications with third-party products . 49
Listener applications 49
Example Listener script 50
Declare Perl API modules and variables for Listener 50
Create and initialize a RIV::Param object for Listener 50
Create and initialize a RIV::App object for Listener 50
Bind the RIV::App object to the message broker subject for Listener 51
Write database records to a log file 51
Send database records to different applications 52

Appendix A. RIV Modules Reference . . 53
RIV module reference 53
RIV module synopsis 53
AddSubject 54
AddTimer 55
DebugLevel 56
DecryptPassword 56
EncryptPassword 57
Latency. 57
MessageLevel. 58
PostInput 59
PublishMessage 59
PublishMessage 60
RetryLimit. 61
RIV::GetInput. 62
RIV::GetResult 63
RIV::InputFilter 65
RIV::InputQueueLength 66
RIV::IsIpNotLoopBackOrMulticast 66
RIV::IsIpValid. 67
RIV::IsIpv4Valid 68
RIV::IsIpv6Valid 68
RIV::ReadDir 69
RIV::RivDebug 70
RIV::RivError 70

RIV::RivMessage 71
RIV::Agent module reference 71
RIV::Agent module synopsis 72
RIV::Agent Constructor 73
GetDNSAllIpAddrs 74
GetDNSAllNames 74
GetDNSFirstIpAddr 75
GetDNSFirstName 76
GetIpArp 77
GetMacArp 77
GetMultTelnet 78
GetPingIP 79
GetPingList 80
GetPingSubnet 80
GetTelnet 81
GetTelnetCols. 82
GetTraceRoute 83
LockThreads 83
PingIP 84
PingList 85
PingSubnet 85
SendNEToDisco 86
SendNEToNextPhase 87
SnmpGet 91
SnmpGetBulk. 92
SnmpGetNext 93
UnLockThreads 94
RIV::App module reference 94
RIV::App module synopsis 94
RIV::App Constructor 95
RIV::OQL module reference 96
RIV::OQL module synopsis 96
RIV::OQL Constructor 97
CreateDB 98
CreateTable 98
Delete 99
Insert 100
Print 102
Select 102
Send 104
Update 104
RIV::Param module reference 106
RIV::Param module synopsis 106
RIV::Param Constructor 106
CommandName 110
DomainName 111
Usage 112
RIV::Record module reference 113
RIV::Record module synopsis 113
RIV::Record Constructor 113
AddLocalNeighbour 114
AddLocalNeighbourTag 115
AddRemoteNeighbour 115
AddRemoteNeighbourTag 116
GetLocalNeighbours 117
GetRemoteNeighbours 117
Print 118
RIV::RecordCache module reference 118
RIV::RecordCache module synopsis 118
RIV::RecordCache Constructor 119
CacheRecord 120
GetRecord 120
GetRecords 121
RIV::SnmpAccess module reference 122
RIV::SnmpAccess module synopsis 122
RIV::SnmpAccess Constructor 123
ASN1ToOid 123
AsyncSnmpGet 124
AsyncSnmpGetBulk 125
AsyncSnmpGetNext 127
GetMibHash. 128
MaxAsyncConcurrent 128
OidToASN1 129
SnmpGet 129
SnmpGetBulk 130
SnmpGetNext 131
SplitOidAndIndex 132

Appendix B. NCP Modules Reference 135
NCP::DBI_Factory module reference. 135
NCP::DBI_Factory module synopsis 135
createDbHandle 137
describeTable 142
execute_insert_auto_inc 144
extractCmdLineOptions 145
extractHashRefOptions 147
insert_auto_inc_row 149
insert_row 150
prepare_insert_auto_inc 152
schema 153
setLogHandle 154
setLogLevel 155
tables 156
timeStamp 158
toUpper 159
NCP::Domain Reference 161
NCP::Domain module synopsis 161
NCP::Domain Constructor 162
clone 164
create 164
drop 166
id 168
name 169
setLogHandle 171
setLogLevel 172

Appendix C. Network Manager glossary 175

Notices 179
Trademarks 181

Index 183

About this publication

IBM Tivoli Network Manager IP Edition provides detailed network discovery, device monitoring, topology visualization, and root cause analysis (RCA) capabilities. Network Manager can be extensively customized and configured to manage different networks. Network Manager also provides extensive reporting features, and integration with other IBM products, such as IBM Tivoli Application Dependency Discovery Manager, IBM Tivoli Business Service Manager and IBM Systems Director.

The *IBM Tivoli Network Manager IP Edition Perl API Guide* describes the Perl API used by third-party developers and technical services personnel to create discovery agents and other client/server applications. These applications can perform such tasks as accessing and modifying records in Network Manager IP Edition databases, and retrieving SNMP information from a network device. This publication is for advanced users who need to customize the operation of Network Manager IP Edition.

Intended audience

This publication is intended for third-party developers and technical services personnel who want to use the Perl API to create discovery agents and other client/server applications.

This publication assumes that you understand how to program in Perl and that you are familiar with IBM Tivoli Network Manager IP Edition.

IBM Tivoli Network Manager IP Edition works in conjunction with IBM Tivoli Netcool/OMNIbus; this publication assumes that you understand how Tivoli Netcool/OMNIbus works. For more information on Tivoli Netcool/OMNIbus, see the publications described in "Publications" on page vi.

What this publication contains

This publication contains the following sections:

- Chapter 1, "Overview of the Perl API," on page 1

  Provides an overview and functional descriptions of all the modules that the Perl API provides.
- Chapter 2, "Writing discovery agents," on page 19

  Describes how to write discovery agents using the `RIV::Agent` and `RIV::Record` modules. Use the example discovery agent scripts as models for writing your own custom discovery agents.
- Chapter 3, "Accessing component databases," on page 35

  Describes how to access and perform actions on Network Manager IP Edition databases using the `RIV::OQL` module. Use the example script as a model for writing your own scripts to access and perform actions on the databases.
- Chapter 4, "Performing SNMP queries," on page 41

  Describes how to retrieve SNMP information from a network device using the `RIV::SnmpAccess` module. Use the example script as a model for writing your own scripts to retrieve SNMP information from a network device.

- Chapter 5, "Writing and integrating Perl applications with third-party products," on page 49

  Describes how to integrate a third party product using the `RIV`, `RIV::Param`, and `RIV::App` modules to interface with Network Manager IP Edition. A sample listener script is provided as a model for writing listener applications.
- Appendix A, "RIV Modules Reference," on page 53

  Provides reference (man) pages for the functions, variables, constants, methods, and virtual methods that the RIV Perl modules provide.
- Appendix B, "NCP Modules Reference," on page 135

  Provides reference (man) pages for the methods that the NCP Perl modules provide.
- Appendix C, "Network Manager glossary," on page 175

  Provides terminology relevant to the Network Manager product.

Publications

This section lists publications in the Network Manager library and related documents. The section also describes how to access Tivoli publications online and how to order Tivoli publications.

Your Network Manager library

The following documents are available in the Network Manager library:

- *IBM Tivoli Network Manager IP Edition Release Notes*, GI11-9354-00

  Gives important and late-breaking information about IBM Tivoli Network Manager IP Edition. This publication is for deployers and administrators, and should be read first.
- *IBM Tivoli Network Manager Getting Started Guide*, GI11-9353-00

  Describes how to set up IBM Tivoli Network Manager IP Edition after you have installed the product. This guide describes how to start the product, make sure it is running correctly, and discover the network. Getting a good network discovery is central to using Network Manager IP Edition successfully. This guide describes how to configure and monitor a first discovery, verify the results of the discovery, configure a production discovery, and how to keep the network topology up to date. Once you have an up-to-date network topology, this guide describes how to make the network topology available to Network Operators, and how to monitor the network. The essential tasks are covered in this short guide, with references to the more detailed, optional, or advanced tasks and reference material in the rest of the documentation set.
- *IBM Tivoli Network Manager IP Edition Product Overview*, GC27-2759-00

  Gives an overview of IBM Tivoli Network Manager IP Edition. It describes the product architecture, components and functionality. This publication is for anyone interested in IBM Tivoli Network Manager IP Edition.
- *IBM Tivoli Network Manager IP Edition Installation and Configuration Guide*, SC27-2760-00

  Describes how to install IBM Tivoli Network Manager IP Edition. It also describes necessary and optional post-installation configuration tasks. This publication is for administrators who need to install and set up IBM Tivoli Network Manager IP Edition.
- *IBM Tivoli Network Manager IP Edition Administration Guide*, SC27-2761-00

  Describes administration tasks for IBM Tivoli Network Manager IP Edition, such as how to administer processes, query databases and start and stop the product.

This publication is for administrators who are responsible for the maintenance and availability of IBM Tivoli Network Manager IP Edition.

- *IBM Tivoli Network Manager IP Edition Discovery Guide*, SC27-2762-00

  Describes how to use IBM Tivoli Network Manager IP Edition to discover your network. This publication is for administrators who are responsible for configuring and running network discovery.
- *IBM Tivoli Network Manager IP Edition Event Management Guide*, SC27-2763-00

  Describes how to use IBM Tivoli Network Manager IP Edition to poll network devices, to configure the enrichment of events from network devices, and to manage plug-ins to the Tivoli Netcool/OMNIbus Event Gateway, including configuration of the RCA plug-in for root-cause analysis purposes. This publication is for administrators who are responsible for configuring and running network polling, event enrichment, root-cause analysis, and Event Gateway plug-ins.
- *IBM Tivoli Network Manager IP Edition Network Troubleshooting Guide*, GC27-2765-00

  Describes how to use IBM Tivoli Network Manager IP Edition to troubleshoot network problems identified by the product. This publication is for network operators who are responsible for identifying or resolving network problems.
- *IBM Tivoli Network Manager IP Edition Network Visualization Setup Guide*, SC27-2764-00

  Describes how to configure the IBM Tivoli Network Manager IP Edition network visualization tools to give your network operators a customized working environment. This publication is for product administrators or team leaders who are responsible for facilitating the work of network operators.
- *IBM Tivoli Network Manager IP Edition Management Database Reference*, SC27-2767-00

  Describes the schemas of the component databases in IBM Tivoli Network Manager IP Edition. This publication is for advanced users who need to query the component databases directly.
- *IBM Tivoli Network Manager IP Edition Topology Database Reference*, SC27-2766-00

  Describes the schemas of the database used for storing topology data in IBM Tivoli Network Manager IP Edition. This publication is for advanced users who need to query the topology database directly.
- *IBM Tivoli Network Manager IP Edition Language Reference*, SC27-2768-00

  Describes the system languages used by IBM Tivoli Network Manager IP Edition, such as the Stitcher language, and the Object Query Language. This publication is for advanced users who need to customize the operation of IBM Tivoli Network Manager IP Edition.
- *IBM Tivoli Network Manager IP Edition Perl API Guide*, SC27-2769-00

  Describes the Perl modules that allow developers to write custom applications that interact with the IBM Tivoli Network Manager IP Edition. Examples of custom applications that developers can write include Polling and Discovery Agents. This publication is for advanced Perl developers who need to write such custom applications.
- *IBM Tivoli Monitoring for Tivoli Network Manager IP User's Guide*, SC27-2770-00

  Provides information about installing and using IBM Tivoli Monitoring for IBM Tivoli Network Manager IP Edition. This publication is for system administrators who install and use IBM Tivoli Monitoring for IBM Tivoli Network Manager IP Edition to monitor and manage IBM Tivoli Network Manager IP Edition resources.

Prerequisite publications

To use the information in this publication effectively, you must have some prerequisite knowledge, which you can obtain from the following publications:

- *IBM Tivoli Netcool/OMNIbus Installation and Deployment Guide*, SC23-9680

  Includes installation and upgrade procedures for Tivoli Netcool/OMNIbus, and describes how to configure security and component communications. The publication also includes examples of Tivoli Netcool/OMNIbus architectures and describes how to implement them.
- *IBM Tivoli Netcool/OMNIbus User's Guide*, SC23-9683

  Provides an overview of the desktop tools and describes the operator tasks related to event management using these tools.
- *IBM Tivoli Netcool/OMNIbus Administration Guide*, SC23-9681

  Describes how to perform administrative tasks using the Tivoli Netcool/OMNIbus Administrator GUI, command-line tools, and process control. The publication also contains descriptions and examples of ObjectServer SQL syntax and automations.
- *IBM Tivoli Netcool/OMNIbus Probe and Gateway Guide*, SC23-9684

  Contains introductory and reference information about probes and gateways, including probe rules file syntax and gateway commands.
- *IBM Tivoli Netcool/OMNIbus Web GUI Administration and User's Guide* SC23-9682

  Describes how to perform administrative and event visualization tasks using the Tivoli Netcool/OMNIbus Web GUI.

Accessing terminology online

The IBM Terminology Web site consolidates the terminology from IBM product libraries in one convenient location. You can access the Terminology Web site at the following Web address:

http://www.ibm.com/software/globalization/terminology

Accessing publications online

IBM posts publications for this and all other Tivoli products, as they become available and whenever they are updated, to the Tivoli Information Center Web site at:

http://publib.boulder.ibm.com/infocenter/tivihelp/v3r1/index.jsp

Note: If you print PDF documents on other than letter-sized paper, set the option in the **File** > **Print** window that allows your PDF reading application to print letter-sized pages on your local paper.

Ordering publications

You can order many Tivoli publications online at the following Web site:

http://www.elink.ibmlink.ibm.com/publications/servlet/pbi.wss

You can also order by telephone by calling one of these numbers:

- In the United States: 800-879-2755
- In Canada: 800-426-4968

In other countries, contact your software account representative to order Tivoli publications. To locate the telephone number of your local representative, perform the following steps:

1. Go to the following Web site:

   http://www.elink.ibmlink.ibm.com/publications/servlet/pbi.wss
2. Select your country from the list and click **Go**. The Welcome to the IBM Publications Center page is displayed for your country.
3. On the left side of the page, click **About this site** to see an information page that includes the telephone number of your local representative.

Accessibility

Accessibility features help users with a physical disability, such as restricted mobility or limited vision, to use software products successfully.

Accessibility features

The following list includes the major accessibility features in Network Manager:

- The console-based installer supports keyboard-only operation.
- The console-based installer supports screen reader use.
- Network Manager provides the following features suitable for low vision users:
  - All non-text content used in the GUI has associated alternative text.
  - Low-vision users can adjust the system display settings, including high contrast mode, and can control the font sizes using the browser settings.
  - Color is not used as the only visual means of conveying information, indicating an action, prompting a response, or distinguishing a visual element.
- Network Manager provides the following features suitable for photosensitive epileptic users:
  - Web pages do not contain anything that flashes more than two times in any one second period.

The Network Manager Information Center, and its related publications, are accessibility-enabled. The accessibility features of the information center are described in Accessibility and keyboard shortcuts in the information center.

Extra steps to configure Internet Explorer for accessibility

If you are using Internet Explorer as your web browser, you might need to perform extra configuration steps to enable accessibility features.

To enable high contrast mode, complete the following steps:

1. Click **Tools** > **Internet Options** > **Accessibility**.
2. Select all the check boxes in the Formatting section.

If clicking **View** > **Text Size** > **Largest** does not increase the font size, click **Ctrl +** and **Ctrl -**.

IBM® and accessibility

See the IBM Human Ability and Accessibility Center for more information about the commitment that IBM has to accessibility.

Tivoli® technical training

For Tivoli technical training information, refer to the following IBM Tivoli Education Web site:

http://www.ibm.com/software/tivoli/education

Support information

If you have a problem with your IBM software, you want to resolve it quickly. IBM provides the following ways for you to obtain the support you need:

Online
: Go to the IBM Software Support site at http://www.ibm.com/software/support/probsub.html and follow the instructions.

IBM Support Assistant
: The IBM Support Assistant (ISA) is a free local software serviceability workbench that helps you resolve questions and problems with IBM software products. The ISA provides quick access to support-related information and serviceability tools for problem determination. To install the ISA software, go to http://www.ibm.com/software/support/isa

Conventions used in this publication

This publication uses several conventions for special terms and actions and operating system-dependent commands and paths.

Typeface conventions

This publication uses the following typeface conventions:

Bold

- Lowercase commands and mixed case commands that are otherwise difficult to distinguish from surrounding text
- Interface controls (check boxes, push buttons, radio buttons, spin buttons, fields, folders, icons, list boxes, items inside list boxes, multicolumn lists, containers, menu choices, menu names, tabs, property sheets), labels (such as **Tip:** and **Operating system considerations:**)
- Keywords and parameters in text

Italic

- Citations (examples: titles of publications, diskettes, and CDs)
- Words defined in text (example: a nonswitched line is called a *point-to-point* line)
- Emphasis of words and letters (words as words example: "Use the word *that* to introduce a restrictive clause."; letters as letters example: "The LUN address must start with the letter *L*.")
- New terms in text (except in a definition list): a *view* is a frame in a workspace that contains data
- Variables and values you must provide: ... where *myname* represents....

`Monospace`

- Examples and code examples

- File names, programming keywords, and other elements that are difficult to distinguish from surrounding text
- Message text and prompts addressed to the user
- Text that the user must type
- Values for arguments or command options

Operating system-dependent variables and paths

This publication uses environment variables without platform-specific prefixes and suffixes, unless the command applies only to specific platforms. For example, the directory where the Network Manager core components are installed is represented as NCHOME.

When using the Windows command line, preface and suffix environment variables with the percentage sign **%**, and replace each forward slash (**/**) with a backslash (****) in directory paths. For example, on Windows systems, NCHOME is %NCHOME%.

On UNIX systems, preface environment variables with the dollar sign **$**. For example, on UNIX, NCHOME is $NCHOME.

The names of environment variables are not always the same in the Windows and UNIX environments. For example, %TEMP% in Windows environments is equivalent to $TMPDIR in UNIX environments. If you are using the bash shell on a Windows system, you can use the UNIX conventions.

Chapter 1. Overview of the Perl API

The Perl API provides developers with the functionality to write discovery agents and other client/server applications. These applications can perform such tasks as accessing and modifying records in Network Manager databases, and retrieving SNMP information from a network device. Developers can also integrate third-party products using the Perl API as a tool to interface with Network Manager.

RIV module overview

The RIV module provides a variable, functions, and virtual methods that the Perl API application modules — `RIV::Agent` and `RIV::App` — use.

Perl API modules used with the RIV module

The following table identifies and briefly describes the Perl API modules used with the `RIV` module:

Perl API Module	Description
`RIV::Agent`	Provides an interface for implementing Network Manager discovery agents.
`RIV::App`	Provides an interface for implementing other Network Manager client/server applications.
`RIV::OQL`	Provides an interface to communicate and perform operations on internal Network Manager databases.
`RIV::Param`	Provides an interface for parsing standard and Network Manager application-specific command line arguments.
`RIV::Record`	Provides a data structure to store the network entity. Typically, you use this data structure in conjunction with the `RIV::Agent` module to write discovery agents.
`RIV::RecordCache`	Provides an interface to access records that reside in a cache.
`RIV::SnmpAccess`	Provides an interface to perform SNMP-related operations on Network Manager MIB trees. **Note:** Discovery agents in previous versions of the Perl API used this module to obtain SNMP information from network devices. Discovery agents implemented with this version of the Perl API should use the SNMP methods that the `RIV::Agent` module provides.

Types of applications

There are two types of applications that you can write using the Perl API:

- Discovery agents — Use the `RIV::Agent` constructor and the ncp_disco_perl_agent binary to create discovery agent applications.
- Other client/server applications — Use the `RIV::App` constructor and the ncp_perl binary to other client/server applications. Examples of these other client/server applications include those that access Network Manager databases.

These application objects are required for interaction with Network Manager IP Edition components (through the virtual methods exported through the `RIV` module) and for instantiation of the other `RIV` modules. Application objects that the `RIV::Agent` and `RIV::App` constructors return are identical for the purpose of accessing other module functionality (for example, `RIV::OQL`).

RIV module functions

The following table identifies and briefly describes the functions that the `RIV` module provides for Network Manager discovery agents and other Network Manager client/server applications:

RIV module function	Description
`RIV::GetInput`	This function has been deprecated. Use the `RIV::GetResult` function.
`RIV::GetResult`	Obtains input either directly or indirectly from message broker.
`RIV::InputFilter`	Binds the specified input function to input tags that match the specified regular expression.
`RIV::InputQueueLength`	Returns the number of items waiting in the application's input queue.
`RIV::IsIpNotLoopBackOrMulticast`	Determines whether the specified address is a valid IP address and not a loop back or multicast address.
`RIV::IsIpValid`	Determines whether the specified address is a valid IP address.
`RIV::IsIpv4Valid`	Determines whether the specified address is a valid IPv4 address.
`RIV::IsIpv6Valid`	Determines whether the specified address is a valid IPv6 address.
`RIV::ReadDir`	Returns a reference to an array of filenames contained in the specified directory.
`RIV::RivDebug`	Prints a list of debug message strings to the standard output.
`RIV::RivMessage`	Prints a list of log message strings to the standard output.
`RIV::RivError`	Displays error messages.

See “RIV module reference” on page 53 for the reference (man) pages associated with these functions.

RIV module virtual methods

The following table identifies and briefly describes the virtual methods that the `RIV` module provides for Network Manager discovery agents and other Network Manager client/server applications:

RIV module virtual method	Description
`AddSubject`	Binds the application to the specified message broker subject.
`AddTimer`	Creates a single-shot or repeating timer.
`DebugLevel`	Provides access to the global Network Manager debug setting through the `RIV::DebugLevel` variable.
`DecryptPassword`	Decrypts a password that was previously encrypted in a previous call to the `EncryptPassword` RIV module virtual method.
`EncryptPassword`	Returns an encrypted representation of the specified password.
`Latency`	Retrieves the timeout for queries.
`PostInput`	Adds a message to the queue.
`PublishMessage`	Publishes the specified message string.
`PublishMessage`	Encodes the hash reference into a message broker string.
`RetryLimit`	Sets the retry limit for queries or returns the maximum number of retries for queries.

See "RIV module reference" on page 53 for the reference (man) pages associated with these functions.

RIV::Agent module overview

The `RIV::Agent` module provides an interface for implementing Network Manager discovery agents. A discovery agent is a specialized application that retrieves connectivity-related information for network entities.

RIV::Agent constructor

The `RIV::Agent` module provides a constructor that creates a discovery agent application object. Use this application object to:

- Interact with Network Manager IP Edition core components libraries using the virtual methods exported from the `RIV` module.
- Instantiate objects for and interact with the other Perl modules: `RIV::Param`, `RIV::Record`, and `RIV::RecordCache`.

Input data records

Input data records that the discovery service sends are supplied through the `RIV::GetResult` method. These input data records can be stored as `RIV::Record` objects, which are nested hash lists to which you can add local and remote neighbors.

Note: All input data records that other services (including the OQL service) send are also supplied through the `RIV::GetResult` method.

Discovery agents and multiple threads

The `RIV::Agent` module allows you to implement discovery agents using multiple threads. The threads implementation creates a single master Perl interpreter that gets copied, one for each thread. Thus, if the discovery agent makes use of three threads, there will be three copies of the master interpreter. Specifically, the `RIV::Agent` module provides the `LockThreads` and `UnLockThreads` methods related to discovery agents and multiple threads.

SNMP operation methods

The `RIV::Agent` module provides methods that discovery agents use to obtain Simple Network Management Protocol (SNMP) information from network devices. These methods obtain this information through the Helper Server. Thus, the Helper Server (and ncp_ctrl) must be running so that the SNMP-related methods can make the appropriate SNMP requests.

The following table identifies and briefly describes the SNMP operation methods that the `RIV::Agent` module provides:

SNMP method	**Description**
`SnmpGet`	Performs an SNMP `get` operation.
`SnmpGetNext`	Performs an SNMP `get-next` operation.
`SnmpGetBulk`	Performs an SNMP `get-bulk` operation.

See "RIV::Agent module reference" on page 71 for the reference (man) pages associated with these methods.

DNS operation methods

The `RIV::Agent` module provides methods that discovery agents use to obtain Domain Name System (DNS) information from network devices. These methods obtain this information through the Helper Server. Thus, the Helper Server (and ncp_ctrl) must be running so that the DNS-related methods can make the appropriate DNS requests.

The following table identifies and briefly describes the DNS operation methods that the `RIV::Agent` module provides:

DNS method	**Description**
`GetDNSAllIpAddrs`	Gets all IP addresses corresponding to a particular node name.
`GetDNSAllNames`	Gets all node names corresponding to the specified IP addresses.
`GetDNSFirstIpAddr`	Gets the first IP address in the list of IP addresses for this node.
`GetDNSFirstName`	Gets the first node name in the list of node names for this IP address.

See “RIV::Agent module reference” on page 71 for the reference (man) pages associated with these methods.

Ping operation methods

The `RIV::Agent` module provides methods that discovery agents use to perform ping operations on network devices. Ping operations determine whether a specific IP or subnet address is accessible. Typically, the ping operation sends a packet to the specified address and waits for a reply.

The `RIV::Agent` module ping operation methods perform the specified ping operation through the Helper Server. Thus, the Helper Server (and ncp_ctrl) must be running so that these ping-related methods can make the appropriate ping requests.

The following table identifies and briefly describes the ping operation methods that the `RIV::Agent` module provides:

ping method	**Description**
`GetPingIP`	Pings the specified IP address and returns whether a network device exists at that address.
`GetPingList`	Pings the specified list of IP addresses and returns a list of network devices that exist at those addresses.
`GetPingSubnet`	Pings the specified subnet and returns whether one or more devices exist at that subnet.
`Ping`	Pings the specified IP address.
`PingList`	Pings the specified list of IP addresses.
`PingSubnet`	Pings the specified subnet.

See “RIV::Agent module reference” on page 71 for the reference (man) pages associated with these methods.

IP and MAC address operation methods

The `RIV::Agent` module provides methods that discovery agents use to perform operations on Internet Protocol (IP) and Medium Access Control (MAC) addresses. The `RIV::Agent` module IP and MAC address operation methods perform the specified address operation through the Helper Server. Thus, the Helper Server (and ncp_ctrl) must be running so that these address-related methods can make the appropriate address operation requests.

The following table identifies and briefly describes the IP and MAC address operation methods that the `RIV::Agent` module provides:

Address method	**Description**
`GetIpArp`	Converts the specified MAC address to an IP address.
`GetMacArp`	Converts the specified IP address to a MAC address.

Address method	Description
`GetTraceRoute`	Traces a route to the specified destination IP address and returns a list of network devices that reside on that route.

See "RIV::Agent module reference" on page 71 for the reference (man) pages associated with these methods.

Telnet operation methods

The `RIV::Agent` module provides methods that discovery agents use to obtain network device information through Telnet rather than SNMP. Like the SNMP methods, the Telnet methods obtain network device-related information through the Helper Server. Thus, the Helper Server (and ncp_ctrl) must be running so that the Telnet-related methods can make the appropriate Telnet requests.

The following table identifies and briefly describes the Telnet operation methods that the `RIV::Agent` module provides:

Telnet method	Description
`GetMultTelnet`	Executes multiple Telnet commands on the specified network device.
`GetTelnet`	Executes the specified Telnet command on the specified network device.
`GetTelnetCols`	Executes the specified Telnet command on the specified network device and splits the return data into table columns.

See "RIV::Agent module reference" on page 71 for the reference (man) pages associated with these methods.

Network entity operation methods

The `RIV::Agent` module provides methods that discovery agents use to perform operations on network entities. Specifically, the `RIV::Agent` module provides methods that perform the following network entity operations:

Network entity method	Description
`SendNEToDisco`	Sends processed records from `RIV::Record` to the returns table of the specified Agent database in Disco.
`SendNEToNextPhase`	Marks the network entity as having completed the current phase and puts the network entity back on the Agent queue ready for processing in the next phase.

See "RIV::Agent module reference" on page 71 for the reference (man) pages associated with these methods.

RIV::App module overview

The `RIV::App` module provides an interface for implementing Network Manager client/server applications within one domain.

RIV::APP constructor

The `RIV::App` module provides two constructors that create a client/server application object. You use this client/server application object to:

- Interact with Network Manager core components libraries using the virtual methods exported from the `RIV` module.
- Instantiate objects for and interact with the other Perl modules: `RIV::OQL`, `RIV::Param`, `RIV::Record`, and `RIV::RecordCache`.

A client/server application can create one or more `RIV::App` application objects as required. For example, two instances of `RIV::App` application objects would be needed in order to implement some special purpose cross-domain behavior.

One example of a client/server application is one that performs one or more OQL queries (in which case the `RIV::OQL` module would also be used).

Note: The `RIV::App` module provides the interface for implementing all Network Manager client/server applications except for discovery agents. To write discovery agents, use the `RIV::Agent` module.

RIV::OQL module overview

The `RIV::OQL` module provides an interface to communicate with and perform operations on Network Manager internal databases.

RIV::OQL constructor

The `RIV::OQL` module provides a constructor that creates and initializes a new `RIV::OQL` object. The `RIV::OQL` constructor takes a blessed reference to either a discovery agent application object or a client/server application object. These application objects were returned in a previous call to the `RIV::Agent` or `RIV::App` constructor. The `RIV::OQL` constructor also takes the name of a service that indicates the Network Manager internal database to use.

Database operation methods

The `RIV::OQL` module provides methods that client/server applications use to perform a variety of operations on Network Manager internal databases.

The following table identifies and briefly describes the database operation methods that the `RIV::OQL` module provides:

Database method	Description
`CreateDB`	Creates a database.
`CreateTable`	Creates a database table.
`Delete`	Deletes records from a database table.
`Insert`	Inserts records into a database table.
`Print`	Prints records as a result of a database query.

Database method	**Description**
`Select`	Executes the specified OQL statement.
`Send`	Communicates with the specified database.
`Update`	Updates records that currently reside in the specified database.

See "RIV::OQL module reference" on page 96 for the reference (man) pages associated with these methods.

RIV::Param module overview

The `RIV::Param` module provides an interface to parse standard and Network Manager application-specific command line arguments.

Standard arguments

Standard arguments are used to specify information about the Network Manager execution environment and to select debug output and application help. All Network Manager applications must support these arguments.

The standard arguments that the `RIV::Param` module provides are summarized in the following table:

Argument	**Description**
`-domain` *domain name*	Specifies the command line argument used to identify the domain in which a user wants to perform some task, for example, starting the Network Manager core components. The *domain name* argument specifies the name of the domain.
`-debug` *debug level*	Specifies the command line argument used to identify the level of debugging output that a user prefers. The *debug level* argument specifies a value from 1-4, where 4 represents the most detailed output.
`-latency` *query latency*	Specifies the command line argument used to specify the maximum time that CLASS waits to connect to another Network Manager process by means of the messaging bus. This option is useful for large and busy networks where the default settings can cause processes to assume that there is a problem when in fact the communication delay is a result of network traffic. The *query latency* argument specifies the maximum time in milliseconds (ms) that CLASS waits.

Argument	Description
`-messagelevel` *message level*	Specifies the command line argument used to identify the level of message output that a user prefers. The *message level* argument specifies the level of message to be logged (the default is warn): • debug • info • warn • error • fatal
`-help`	Specifies the command line argument used to display command line options.

RIV::Param constructor

The `RIV::Param` module provides a constructor that creates and initializes a new `RIV::Param` object. The `RIV::Param` constructor takes three optional parameters used to specify:

- Application-specific parameters
- Usage information or nonstandard command line argument scenarios
- Help information written to standard output

The `RIV::Param` object can be used as the first parameter to the `RIV::App` constructor in place of the domain name argument. In this case, the application object is created with the specified values. Likewise, the `RIV::Param` object can be used as the first parameter to the `RIV::Agent` constructor.

See "RIV::Param Constructor" on page 106 for details.

RIV::Param module constants

The `RIV::Param` module also provides several constants that the `RIV::Param` constructor uses to identify a particular command line as follows:

Constant	Description
`RivParamNoArg`	Specifies that the command line takes no arguments.
`RivParamSingleArg`	Specifies that the command line takes one argument.
`RivParamMandatory`	Specifies that the command line takes a mandatory argument.

Parameter operation methods

The `RIV::Param` module provides methods that client/server applications use to print usage information or to obtain information about domain and command names.

The following table identifies and briefly describes the parameter operation methods that the `RIV::Param` module provides:

Parameter method	**Description**
`CommandName`	Returns the name of the command.
`DomainName`	Returns the name of the domain.
`Usage`	Writes a brief usage explanation to standard output.

See "RIV::Param module reference" on page 106 for the reference (man) pages associated with these methods.

RIV::Record module overview

The `RIV::Record` module provides a data structure to store network entity data records.

Network entities

The `RIV::Record` module is used in conjunction with the `RIV::Agent` module to write discovery agents. The `RIV::Record` data structure stores records associated with the network entities sent by DISCO to the Perl Discovery Agent. You can then add local neighbors and remote neighbors to this record by calling the appropriate local and remote neighbor operation methods.

RIV::Record constructor

Before accessing the methods that the `RIV::Record` module provides, you must call the `RIV::Record` constructor to create and initialize a new `RIV::Record` data structure. This data structure stores network entity records retrieved from DISCO.

RIV::Record data structure

The following respresents a `RIV::Record` data structure:

```
$refLocalNeighbours = $record->{m_LocalNbr};
@LocalNeighbours = @$refLocalNeighbours;
$refRemoteNeighbours = $LocalNeighbours[$i]->{m_RemoteNbr};
@RemoteNeighbours = @$refRemoteNeighbours;
$refRemoteNeighbour = $RemoteNeighbours[$j];
%remoteNeighbour = %$refRemoteNeighbour;
```

The value for the key `m_LocalNbr` is a pointer to an array, which is a list of hashes, where each hash represents a local neighbor. If there are any remote neighbors, the local neighbor has a key (`m_RemoteNbr`) whose value points to the reference of an array, which is a list of hashes, each representing a remote neighbor. You will need this data structure if you intend to manipulate it directly. In most cases, however, your task is limited to creating hash lists that define the local and remote neighbors.

The `AddLocalNeighbour` and `AddRemoteNeighbour` methods can be used to add neighbors, and the `GetLocalNeighbours` and `GetRemoteNeighbours` methods can be used to retrieve information about neighbors.

Local and remote neighbor operation methods

The `RIV::Record` module provides methods that discovery agents use to perform add and get operations on local and remote neighbors.

The following table identifies and briefly describes the local and remote operation methods that the `RIV::Record` module provides:

Local and remote neighbor method	Description
`AddLocalNeighbour`	Adds a local neighbor.
`AddLocalNeighbourTag`	Adds a tag to a local neighbor.
`AddRemoteNeighbour`	Adds a remote neighbor.
`AddRemoteNeighbourTag`	Adds a tag to a remote neighbor.
`GetLocalNeighbours`	Returns an array of local neighbors.
`GetRemoteNeighbours`	Returns an array of remote neighbors.
`Print`	Prints the current record.

See "RIV::Record module reference" on page 113 for the reference (man) pages associated with these methods.

RIV::RecordCache module overview

The `RIV::RecordCache` module provides an interface to access a record cache file.

RIV::RecordCache constructor

Before accessing the methods that the `RIV::RecordCache` module provides, you must call the `RIV::RecordCache` constructor to create and initialize a new record cache file object. The `RIV::RecordCache` constructor takes a blessed reference to either a discovery agent application object or a client/server application object. These application objects were returned in a previous call to the `RIV::Agent` or `RIV::App` constructor.

The `RIV::RecordCache` constructor also takes the name of the record cache file object to be created and an optional path to this object. By default, the optional path is `$NCHOME/var/precision`

Record cache operation methods

The `RIV::RecordCache` module provides methods that applications use to perform a variety of operations on records that reside in the cache file.

The following table identifies and briefly describes the record cache operation methods that the `RIV::RecordCache` module provides:

Record cache method	Description
`CacheRecord`	Adds a record to the cache file.
`GetRecord`	Retrieves a record from the cache file.
`GetRecords`	Retrieves a list of all records residing in the cache file.

See “RIV::RecordCache module reference” on page 118 for the reference (man) pages associated with these methods.

RIV::SnmpAccess module overview

The `RIV::SnmpAccess` module provides an interface to perform SNMP-related operations on Network Manager MIB trees.

Obtaining SNMP information with the RIV::Agent and RIV::SnmpAccess modules

The following list summarizes how discovery agents should deal with obtaining SNMP information with this version of the Perl API:

- The Helper Server (and ncp_ctrl) must be running so that the Get, GetNext, and GetBulk methods provided by the `RIV::Agent` and `RIV::SnmpAccess` modules can make the appropriate queries.
- Discovery agents implemented with this version of the Perl API should use the Get, GetNext, and GetBulk methods provided by the `RIV::Agent` module to obtain SNMP information from a network device.
- Discovery agents implemented with previous versions of the Perl API and that called the Get, GetNext, and GetBulk methods provided by the `RIV::SnmpAccess` module will work. There is no need to port these discovery agents to use the Get, GetNext, and GetBulk methods provided by the `RIV::Agent` module.

RIV::SnmpAccess constructor

Before accessing the methods that the `RIV::SnmpAccess` module provides, you must call the `RIV::SnmpAccess` constructor to create and initialize a new `RIV::SnmpAccess` object. The `RIV::SnmpAccess` constructor takes a blessed reference to either a discovery agent application object or a client/server application object. These application objects were returned in a previous call to the `RIV::Agent` or `RIV::App` constructor.

Maximum number of concurrent asynchronous requests

The `RIV::SnmpAccess` module provides a `MaxAsyncConcurrent` variable that sets the maximum number of concurrent asynchronous requests.

Synchronous and asynchronous SNMP operation methods

The `RIV::SnmpAccess` module provides an interface to the Vertigo SNMP and MIB library functions. Both synchronous and asynchronous variants of each SNMP Get method are provided. The synchronous versions cause the caller to wait until the results are available (or the request has failed). The asynchronous versions all return results via `RIV::GetResult`. By using this latter method, overlapped I/O may be implemented without the complexity of using Perl threads.

SNMP operation methods

The `RIV::SnmpAccess` module provides methods that discovery agents and client/server applications use to perform SNMP operations on a network device through the Helper Server. Thus, the Helper Server (and ncp_ctrl) must be running so that the SNMP-related methods can make the appropriate SNMP requests.

The following table identifies and briefly describes the SNMP operation methods that the `RIV::SnmpAccess` module provides:

SNMP operation method	Description
`ASN1ToOid`	Converts the specified ASN.1 value to its corresponding OID.
`AsyncSnmpGet`	Performs an asynchronous SNMP `get` operation on the specified MIB variable.
`AsyncSnmpGetBulk`	Performs an asynchronous SNMP `get-bulk` operation on all MIB objects in the specified MIB table.
`AsyncSnmpGetNext`	Performs an asynchronous SNMP `get-next` operation on the specified MIB variable.
`GetMibHash`	Gets the entire MIB tree by browsing the files that exist in the `$NCHOME/mibs` directory.
`OidToASN1`	Converts the specified OID to its corresponding ASN.1 value.
`SnmpGet`	Performs a synchronous SNMP `get` operation on the specified MIB variable.
`SnmpGetBulk`	Performs a synchronous SNMP `get-bulk` operation on all MIB objects in the specified MIB table.
`SnmpGetNext`	Performs a synchronous SNMP `get-next` operation on the specified MIB variable.
`SplitOidAndIndex`	Converts the full ASN.1 value into its index and the base OID.

See "RIV::SnmpAccess module reference" on page 122 for the reference (man) pages associated with these methods.

NCP modules overview

The NCP modules provide interfaces that operate on the NCIM topology database and domains.

Summary of Perl API NCP modules

The following table identifies and briefly describes the Perl API NCP modules:

Perl API NCP Module	Description
`NCP::DBI_Factory`	Provides an interface to make it easier to use the standard Perl DBI module to perform operations on the Network Connectivity and Inventory Model (NCIM) topology database.
`NCP::Domain`	Provides an interface to perform operations on NCIM domains.

NCP::DBI_Factory module overview

The `NCP::DBI_Factory` module provides an interface to make it easier to use the standard Perl DBI module to perform operations on the NCIM topology database. Use of this module assumes that you understand the standard Perl DBI module. The `NCP::DBI_Factory` module reads the database login details from DbLogins.*DOMAIN*.cfg, which allows it access to the pre-configured data sources such as NCIM.

DBI handle

The `NCP::DBI_Factory` module provides a method that creates and initializes a new DBI handle. You pass this handle in subsequent calls to the methods that perform operations on the specified NCIM topology database. This DBI handle contains the information needed to connect to the requested NCIM topology database.

Databases that the DBI_Factory module supports

The `NCP::DBI_Factory` module currently supports operations on the following databases:
- DB2®
- Informix
- MySql
- Oracle

For all of these databases, table and field names are case-insensitive from the point of view of SQL statements. However, rows returned by both the DB2 and Oracle databases will have all field names in upper case, rows returned by the MySql database will have all field names in mixed case, and rows returned by Informix will have field names in lower case. The `NCP::DBI_Factory` module provides the `toUpper` method that returns a copy of a single row with all lower case field names in upper case.

NCIM Database operation methods

The `NCP::DBI_Factory` module provides methods that client/server applications use to perform a variety of operations on the specified NCIM topology database.

The following table identifies and briefly describes the database operation methods that the `NCP::DBI_Factory` module provides:

NCIM Database method	Description
`createDbHandle`	Creates a standard DBI handle to be used in subsequent calls to the other `NCP::DBI_Factory` methods.
`describeTable`	Returns a sorted array of upper case field names for the specified table or view.
`extractCmdLineOptions`	Allows database login options for the DBI handles to be provided in a common format.
`extractHashRefOptions`	Extracts database login options from a reference to a hash.
`insert_auto_inc_row`	Inserts a row into a named table, where the table has an auto incremented column.
`insert_row`	Inserts a row into a named table.

NCIM Database method	Description
`schema`	Returns the schema name associated with the NCIM topology database being used.
`setLogHandle`	Passes in a log handle associated with an opened file used for logging messages.
`setLogLevel`	Sets the log level for error and message reporting.
`tables`	Returns a sorted array of table and view names for the current schema.
`timeStamp`	Returns the current timestamp in a format suitable for addition to the NCIM topology database.
`toUpper`	Returns a copy of a hash (a single row retrieved from an NCIM database table) with all field names converted to upper case.

See "NCP::DBI_Factory module reference" on page 135 for the reference (man) pages associated with these methods.

NCP::Domain module overview

The `NCP::Domain` module provides an interface to perform operations on NCIM Network Manager domains.

NCP::Domain constructor

Before accessing the methods that the `NCP::Domain` module provides, you must call the `NCP::Domain` constructor to create a new `NCP::Domain` object. The `NCP::Domain` constructor requires the domain name and options for the database connection. The newly created `NCP::Domain` object encapsulates the attributes associated with the specified domain and database connection.

NCIM domain operation methods

The `NCP::Domain` module provides methods that client/server applications use to perform a variety of operations on a specified NCIM domains. Some of these operations involve the domainMgr table in the NCIM topology database that resides in the specified domain.

The following table identifies and briefly describes the database operation methods that the `NCP::Domain` module provides:

NCIM domain database method	Description
`clone`	Creates a new domain that is a copy of an existing domain.
`create`	Creates an entry in the domainMgr table for this domain if one does not already exist.
`drop`	Removes all references to the specified domain from the domainMgr table.
`id`	Retrieves the domainMgrId from the domainMgr table in the NCIM topology database that resides in the specified domain.

NCIM domain database method	Description
`name`	Returns the domain name for the current domain.
`setLogHandle`	Passes in a log handle associated with an opened file used for logging messages.
`setLogLevel`	Sets the log level for error and message reporting.

See "NCP::Domain Reference" on page 161 for the reference (man) pages associated with these methods.

Synchronization with message broker

The Network Manager IP Edition core components use of message broker is highly multithreaded, whereas Perl applications are single-threaded. Although support for threads was included from Perl 5.005 onwards, this is neither operating system-native, nor POSIX in semantics. Thus, there is no possibility of direct thread-safe integration between the Network Manager IP Edition and Perl code.

All modules contained in `RIV` (except for the `RIV::Agent` module) assume a single Perl thread. If multiple Perl threads are used, a single thread (the one in which the session was instantiated) must be used for interaction with Network Manager IP Edition core components. The `RIV::SnmpAccess` module provides both synchronous and asynchronous methods that allow you to perform operations on the MIB tree. The methods `RIV::InputQueueLength` and `RIV::GetResult` are used to query and extract from the application's input queue.

The `RIV::Agent` module provides a multithreading capability into the Perl discovery agent.

See "Using threads in discovery agents" on page 33 for more information.

Installing the Perl API

The Perl API and its associated modules reside in a specific directory.

After installing the Perl API, the required version of Perl and its associated modules reside in the `$NCHOME/precision/perl` directory.

Note: The Perl API is installed when you install either the IBM Tivoli Monitoring core components or the Web GUI. Typically, you source the appropriate environment variables in `$NCHOME/precision/env.sh` to set up the required environment to use the Perl API.

Perl builds

Network Manager IP Edition provides two customized Perl executables as it is necessary to use static linkage against the Network Manager IP Edition core components libraries.

The first executable is the **ncp_disco_perl_agent** binary. This binary is used by Discovery agents and it is executed automatically by ncp_disco. You would not expect to run this binary directly.

The second executable is the **ncp_perl** binary. You use this binary to develop Perl scripts to customize ITNM or to integrate with other products.

Obtaining SNMP information from a network device

The Perl API allows discovery agents and other client/server applications to obtain SNMP information from a network device through the Helper Server. Typically, this information is retrieved from one or more MIB variables or an entire MIB table.

Helpers retrieve information from the network during a discovery. More specifically, the SNMP helper (ncp_dh_snmp) returns results of an SNMP request such as Get, GetNext, GetBulk, and so forth. The Helper Server can service these SNMP requests directly with cached data or pass on the request to the SNMP helper. The methods that the `RIV::Agent` and `RIV::SnmpAccess` modules provide query the Helper Server. Therefore, the Helper Server must be running so that the methods in these modules can obtain SNMP information from network devices.

Note: It is no longer possible to obtain SNMP information directly from a network device as all queries are now made through the Helper Server.

See the *IBM Tivoli Network Manager IP Edition Administration Guide* (SC27-2761-00) for more information on the Helper Server.

In previous versions of the Perl API, the only way to obtain SNMP information from a network device was to call the methods defined in the `RIV::SnmpAccess` module. Futhermore, a discovery agent could obtain SNMP information directly from a network device.

The following list summarizes how discovery agents and other client/server applications should deal with obtaining SNMP information with this version of the Perl API:

- The Helper Server (and ncp_ctrl) must be running so that the Get, GetNext, and GetBulk methods provided by the `RIV::Agent` and `RIV::SnmpAccess` modules can make the appropriate queries.
- Discovery agents implemented with this version of the Perl API should use the Get, GetNext, and GetBulk methods provided by the `RIV::Agent` module to obtain SNMP information from a network device.
- Discovery agents implemented with previous versions of the Perl API and that called the Get, GetNext, and GetBulk methods provided by the `RIV::SnmpAccess` module will work. There is no need to port these discovery agents to use the Get, GetNext, and GetBulk methods provided by the `RIV::Agent` module.

Perl API modules reference page syntax

The Perl API modules reference pages use a consistent reference page format.

Each Perl API module reference page uses the following format:

- Constructor/Method — This section specifies the name of the constructor or method associated with this Perl API module.
- Synopsis — This section provides the definition for this constructor or method.
- Description — This section provides a description of the functionality that this constructor or method provides.
- Parameters — This section provides descriptions for the input parameters identified in the Synopsis for this constructor or method. If there are no input parameters, this section specifies None.
- Returns — This section provides a description of the value or values that this constructor or method returns. If no values are returned, this section specifies None.
- Notes® — This optional section provides additional information about a constructor or method.
- Example Usage — This section provides an example of how to call this constructor or method.
- See Also — This section provides references to modules or methods that you should be aware of when using this module's constructor or methods.

Chapter 2. Writing discovery agents

Discovery agents retrieve information about devices in the network. They also report on new devices by finding new connections when investigating device connectivity. Discovery agents are used for specialized tasks. For example, the ARP Cache discovery agent populates the Helper Server database with IP address-to-MAC address mappings. You can use the Perl API to write custom discovery agents that perform a variety of useful and specialized tasks, including retrieving information about the connectivity of network entities.

To create custom discovery agents in Perl, you use the `RIV`, `RIV::Agent`, `RIV::Param`, and `RIV::Record` modules.

Before you write a discovery agent

The Perl API is designed to enable the easy creation and prototyping of custom discovery agents. Before writing a custom discovery agent, you need to perform some prerequisite tasks and to be familiar with the discovery process.

Before writing a custom discovery agent, ensure that you have completed these steps:

1. Copied the Agent Definition File (**.agnt** file extension) to the `$NCHOME/precision/disco/agents` directory.

   See"Prototype agent definition file template" on page 31 for details about this file.
2. If additional MIBS are required, ensure that they are copied to the `$NCHOME/precision/mibs` directory and that the ncp_mib process is run to import these MIBs into the database.
3. Created new discovery stitchers to process the returns data from the new discovery agent and to add the data into the topology.

   See the *IBM Tivoli Network Manager IP Edition Language Reference* (SC27-2768-00) for a detailed description of stitchers and the stitcher language.
4. Started the Helper Server (and ncp_ctrl). The Helper Server (and ncp_ctrl) must be running because the SNMP-related methods that the `RIV::Agent` module provides query the Helper Server. It is not possible to obtain SNMP information directly from a network device.

In addition, you should also be familiar with:

- The architecture of the discovery process
- How discovery agents communicate with the DISCO process and the Helper Servers
- The different databases created in the DISCO process to enable discovery agents to work successfully

See *IBM Tivoli Network Manager IP Edition Product Overview* (GC27-2759-00) for more information on the previously listed topics.

Writing a discovery agent

Writing a discovery agent requires you to use the `RIV::Agent` and `RIV::Record` modules and to perform a number of prescribed steps. Many discovery agents also use the `RIV` and `RIV::Param` modules.

The following topics describe the steps to follow when writing a discovery agent, using the IP routing discovery agent as an example.

Step 1: Create an agent.pl file

Create an *agent*`.pl` file to contain the Perl code that implements the discovery agent.

Where:

- *agent* — Specifies the name of the file to contain the discovery agent Perl code. For example: `ASAgent.pl`, `TunnelAgent.pl`, and `NATTextFileAgent.pl` are *agent*`.pl` files that contain the Perl code that implement their respective discovery agents.

Create the *agent*`.pl` file in the `$NCHOME/precision/disco/agents/perlAgents` directory.

Step 2: Declare Perl modules

Declare the Perl modules (using `use` statements) the discovery agent requires. For example:

```
use RIV;
use RIV::Param;
use RIV::Record;
use RIV::Agent;
```

Step 3: Create a new agent

To begin, you must create a new discovery agent using the `RIV::Agent` constructor that the `RIV::Agent` module provides. This constructor sets the name of the discovery agent and also sets up the TCP connections to the DISCO and Helper Server processes. For example:

```
$param = new RIV::Param();
$agent = new RIV::Agent($param, "foo");
```

The example shows that the `RIV::Agent` constructor consists of two parameters:

- *$param* — Specifies a `RIV::Param` object. As the example shows, this object is returned in a call to the `RIV::Param` constructor.
- *$agentName* — Specifies a string that identifies the name of this discovery agent. In this example, the name of the agent is `foo`.

The `RIV::Agent` constructor returns a discovery agent application object to the *$agent* variable. You reference all `RIV::Agent` module methods through this object. For example: `$agent->SnmpGet(...)`, `$agent->SnmpGetNext(...)`, and so forth.

Step 4: Wait for input from the DISCO process

Once the finders detect a network entity that has an OID matching a device that needs to be processed by the discovery agent, the network entity is inserted into the agent's despatch table.

Note: The list of devices supported by the DISCO process is defined in the Agent Definition File.

The DISCO process then sends the record of the device to the agent for processing. This record is received by the Perl discovery agent using the `$agent->RIV::GetResult()` method. The records received from DISCO are tagged with the string `NE`. For example:

```
DEVICE: while (1)
{
    my ($tag,$data) = RIV::GetInput(-1);
    if ($tag ne "NE")
    {
        print "Data is not a network entity. Ignoring it!\n";
        next DEVICE;
    }
my ($tag, $data) = $agent->RIV::GetResult(-1);
if ($tag ne "NE")
{
print "Data is not a network entity. Ignoring it!\n";
next DEVICE;
}
else
{
print "This agent is going to process the device!\n";
}
```

Step 5: Create a RIV::Record object

When DISCO sends a record for processing by the discovery agent, the record can be conveniently stored in a data structure. This data structure is referred to as a `RIV::Record` object. You use the `RIV::Record` constructor that the `RIV::Record` module provides to create this object. For example:

```
my $TestNE = new RIV::Record($data);
```

The `RIV::Record` constructor takes the following parameter:

- *$refNE* — Specifies a reference to a hash list.

The record that DISCO sends for processing may be a request from ncp_disco for the agent to terminate. The following code checks for this termination request:

```
my $TestNE = new RIV::Record($data);

    if ($TestNE->{m_TerminateAgent})
    {
        log_msg("Exit Main Loop\n");
        exit(0);
    }
```

The `RIV::Record` constructor returns a `RIV::Record` object.

The `RIV::Record` module provides methods that enable you to easily add and retrieve local and remote neighbors. For example, the following example shows a call to the `AddLocalNeighbour` method.

```
my %localNbr;
$localNbr{'m_IpAddress'} = '1.2.3.4';
$localNbr{'m_IfIndex'} = 2;
$TestNE->AddLocalNeighbour(\%localNbr);
```

The `AddLocalNeighbour` method takes a *$refNbr* parameter that specifies a reference to a hash list. This hash list defines the local neighbor as a set of key value pairs (*varBinds*).

Step 6: Decide if the agent must process the device (pre-mediation layer)

In the pre-mediation layer you must write Perl code to decide if the device needs to be processed by this discovery agent. The Agent Definition File should be used to filter out devices based on the OIDs.

Note: The list of devices supported by the DISCO process is defined in the Agent Definition File.

Typically, this Perl code checks the device's IP address using the `RIV::IsIpValid` method, as shown in the following example:

```
print "Checking if IP address is valid..\n";
if (!RIV::IsIpValid($host))
{
print "Device has invalid IP address. Ignoring the record!\n";
next DEVICE;
}
```

If the device's IP address is valid, then the discovery agent processes the device. Otherwise, the discovery agent does not process the device. In the example an appropriate message would display to standard output if the device has an invalid IP address.

Step 7: Retrieve device information from the Helper Server (meditation layer)

Next, you must retrieve device information from the Helper Server. This can be achieved using SNMP Get, Telnet, or DNS. For example:

```
{
$refLifindex=$agent->SnmpGetNext($TestNE,'ipAdEntIfIndex');
$refLnetmask=$agent->SnmpGetNext($TestNE,'ipAdEntNetMask');
$refLphysaddress=$agent->SnmpGetNext($TestNE,'ifPhysAddress');
$refRifindex=$agent->SnmpGetNext($TestNE,'ipRouteIfIndex');
$refRtype=$agent->SnmpGetNext($TestNE,'ipRouteType');
$refRnexthop=$agent->SnmpGetNext($TestNE,'ipRouteNextHop');
}
```

The above example uses the `RIV::Agent` method `SNMPGetNext`. The `SNMPGetNext` method takes two parameters:

- *$ne* — Specifies the network entity, which is typically a `RIV::Record` object.
- *$oid* — Specifies a MIB variable, for example, `ipAdEntIfIndex` in the above example.

The above example performs SNMP GET operations on the network entity (NE) in question and retrieves the specified MIB variables. If you prefer, you could substitute the `SnmpGetNext` method with the appropriate methods to allow Telnet or DNS access.

Step 8: Determine local and remote neighbors (processing layer)

In the processing layer, the local and remote neighbors are determined, based on the information from the Mediation layer. A local neighbor is a network interface that resides on the device being discovered. A remote neighbor is something connected to one of these network interfaces.

The following example is taken from the IP routing discovery agent:

```
sub Processing
{
 print "Processing the local neighbours\n";
 foreach $entry (@$refLifindex){
  if (RIV::IsIpValid($entry->{ASN1})){
   my %localNbr;
   $localNbr{'m_IpAddress'} = $entry->{ASN1};
   $localNbr{'m_IfIndex'} = $entry->{VALUE};
   $TestNE->AddLocalNeighbour(\%localNbr);
 }
 }
}
```

Step 9: Sending the processed record to DISCO

Once the network entity has been processed, its record needs to be sent to DISCO. The `RIV::Agent` method `SendNEToDisco` allows you to accomplish this task. The `SendNEToDisco` method takes two parameters:

- *$entity* — Specifies a reference to a hash list that contains the definition of the record to be sent to DISCO. For convenience, the `RIV::Record` module provides an object that serves as a hash list with nested structures for representing local and remote neighbors.
- *$lastRecTag* — Specify the value 1 to indicate that this is the last record for the network entity. Specify the value 0 (zero) to indicate that more records for this network entity are to follow.

  If you use `RIV::Record` objects, the `SendNEToDisco` method ignores this parameter.

Step 10: Running the newly created agent

Before running the newly created agent, make sure that you have:

- Created the agent definition file (*agentName*`.agnt`) in the `$NCHOME/precision/disco/agents` directory. The following example shows the agent definition file for a discovery agent called `CustomPerlAgent`:

  ```
  $NCHOME/precision/disco/agents/CustomPerlAgent.agnt
  ```
- Created or copied the Perl discovery agent script (*agentName*`.pl`) in the `$NCHOME/precision/disco/agents/perlAgents` directory. The following example shows the Perl discovery agent script for a discovery agent called `CustomPerlAgent`:

  ```
  $NCHOME/precision/disco/agents/perlAgents/CustomPerlAgent.pl
  ```
- Registered the agent with ITNM using the following command:

  ```
  $NCHOME/precision/bin/ncp_agent_registrar -register agentName
  ```

  Where: *agentName* specifies the name of the discovery agent.

  The following example shows how to register a discovery agent called `CustomPerlAgent`:

  ```
  $NCHOME/precision/bin/ncp_agent_registrar -register
  CustomPerlAgent
  ```

Once the agent has been registered, you should be able to see it and any other registered discovery agents in the Discovery Configuation GUI. Use the Discovery Agent GUI to enable the discovery agent for the next discovery.

For information on the Discovery Configuation GUI, see *IBM Tivoli Network Manager IP Edition Discovery Guide* (SC27-2762-00).

Example discovery agents

Typically, a network environment contains multiple discovery agents to support the wide variety of network devices operating in this environment. Thus, the types of discovery agents you can implement with the Perl API is extensive.

The following topics provide simple examples of discovery agents and a skeleton outline of a discovery agent that you can use as a template.

Discovery agent skeleton

The discovery agent skeleton provides an outline of the sections typically implemented in a discovery agent that makes use of the Perl API. This outline also specifies calls to the constructors and methods (for example, `new RIV::Agent`, `RIV::IsIpValid`, `RIV::Agent::SendNEToDisco`, and so forth) typically used in a discovery agent. Use the discovery agent skeleton as a way to start implementing your custom discovery agents.

The following Perl script provides a skeleton outline for a simple discovery agent. Explanations of specific lines follow the skeleton outline:

```
use RIV;
use RIV::Param;
use RIV::Record;
use RIV::Agent;  1
# Create a new discovery agent
print "Creating a new agent\n";  2

sub Init{
 my $param=new RIV::Param();
   $agent=new RIV::Agent($param, "PerlDetails");  3
 # Wait for input from the DISCO process
   print "Entering infinite loop wait for devices for Disco\n";  4

   DEVICE: while (1){
    my ($tag, $data) = $agent->RIV::GetResult(-1);
    if ($tag ne 'NE'){  5
    print "Data is not a Network entity Ignoring it!\n";
    next DEVICE;
 }
# Create a RIV::Record object
my $TestNE = new RIV::Record($data);

    if ($TestNE->{m_TerminateAgent})
    {
        log_msg("Exit Main Loop\n");
        exit(0);
    }  6

# Decide if the agent must process the device (pre-mediation layer)
  ...
  ...
  print "Checking if IP address valid..\n";

  if (!RIV::IsIpValid($host)){  7
   print "Device has invalid IP address ignoring the record!\n";
  next DEVICE;
 }
 # Retrieve device information from the Helper Server (mediation layer)
  ...
 print "Entering Mediation layer\n";  8
 Mediation();

 print "Entering Processing layer\n";
 Processing();  9

 print "Sending Record to Discovery\n";
 $agent->SendNEToDisco($TestNE,0);  10
 }

sub Mediation{  11
   . . .
   . . .
   # Retrieve the relevant information from the Helper Server using SNMP,
Telnet or DNS.
   . . .
   . . .
 }
 sub Processing{  12
   . . .
   . . .
   # Determine local and remote neighbors (processing layer) based
   # on the information retrieved in the Mediation Layer.
   # The neighbours are then added to the RIV::Record representing
   # the network entity.
   . . .
   . . .
 }
 Init();
```

The following list explains specific numbered items in the previously listed skeleton outline of a discovery agent:

1. Declare the Perl API modules to use in the discovery agent. The `RIV::Agent` and `RIV::Record` modules are required. The optional `RIV::Param` module is useful for parsing standard and application-specific command line arguments.
2. Calls the `print` operator to display a message to the standard output indicating the creation of a new discovery agent.
3. This is the create or initiate discovery agent section. Creates a new discovery agent with the specified name. The skeleton outline specifies a discovery agent with the name of `PerlDetails`.
4. The discovery agent is ready to receive records from DISCO.
5. Checks that the data records received from DISCO have been tagged with the string `NE`.
6. Handles a request from ncp_disco to terminate the Perl discovery agent if the test evaluates to true.
7. Checks that the device has a valid IP address.
8. This is the mediation layer section. Gets the relevant SNMP information.
9. This is the processing layer section. Interprets the information to find the local and remote neighbors.
10. Sends the record and filled-out network entity to DISCO.
11. Implements the `Mediation` method.
12. Implements the `Processing` method.

Network entity discovery agent example

The purpose of many discovery agents is to accept network entities from DISCO, process these entities in some way, and then return the result. This example network entity discovery agent provides a skeleton outline for these tasks. Use this example discovery agent skeleton to write discovery agents that need to accomplish these tasks.

The following Perl script provides a skeleton outline for a simple network entity discovery agent. Explanations of specific lines follow the skeleton outline:

```
1 use RIV;
2 use RIV::Param;
3 use RIV::Agent;
4 use RIV::Record;
5
6 $param = RIV::Param::new();
7 $agent = RIV::Agent::new($param, "PerlAgent");
8
9 while (1)
10 {
11   my ($tag, $data) = $agent->RIV::GetResult(-1);
12   next unless ($tag eq "NE");
13
14   foreach my $ne (@{ $data })
15   (
16    $NE = RIV::Record::new($ne);
17    ...
18    ...
19    $agent->SendToDisco($$ne,1);
20   )
21 )
```

The following list explains specific numbered items in the previously listed discovery agent example:

- **Lines 1-4**

  Declare the Perl API modules to use in the discovery agent. The `RIV::Agent` and `RIV::Record` modules are required. The optional `RIV::Param` module is useful for parsing standard and application-specific command line arguments.
- **Lines 14-16**

  Checks that the data received from DISCO has been tagged with the string `NE`.
- **Line 19**

  Returns the data to DISCO.

IP routing discovery agent example

The IP routing discovery agent Perl program shows how a simple discovery agent can be written using the Perl API. Study this example to acquire additional knowledge about how to write discovery agents using the Perl API.

The following IP routing discovery agent example uses a representative number of the methods provided in the `RIV::Agent`, `RIV::Record`, and `RIV::Param` Perl API modules. Explanations of specific lines follow the program.

```
use RIV;
use RIV::Param;
use RIV::Record;
use RIV::Agent;  1

print "Creating a new agent\n";  2
Init();
```

```
print "Entering infinite loop wait for devices for DISCO\n"; 3

 DEVICE: while (1){ 4
  my ($tag, $data) = $agent->RIV::GetResult(-1); 5
  if ($tag ne 'NE'){ 6
     print "Data is not a Network entityIgnoring it!\n";
       next DEVICE; 6
  }

  my $TestNE = new RIV::Record($data);
    if ($TestNE->{m_TerminateAgent})
    {
        log_msg("Exit Main Loop\n");
        exit(0); 7
    }
  $TestNE->{'m_LastRecord'}=1; 8
  $TestNE->{'m_UpdAgent'}="PerlDetails"; 9
  my $host=$TestNE->{'m_IpAddress'}; 10
```

The following list explains specific numbered items in the previously listed IP routing discovery agent Perl program:

1. Declare the Perl API modules to use in this discovery agent. The `RIV::Agent` and `RIV::Record` modules are required. The optional `RIV::Param` module is useful for parsing standard and application-specific command line arguments.
2. Calls the `print` operator to display a message to the standard output indicating the creation of a new discovery agent.
3. Calls the `print` operator to display a message to the standard output indicating the program is ready to receive records from the DISCO process.
4. Sets up an infinite loop waiting to receive data from DISCO.
5. Calls the `RIV::GetResult` function to return a data record from DISCO. In this call, the value `-1` is passed signifying that `RIV::GetResult` should "wait forever" to received data records from DISCO before returning.
6. Checks the data record that `RIV::GetResult` returns to the *$tag* variable. If the returned data record has not been tagged with the string `NE`, then use the `print` operator to display a message to the standard output indicating this data record is not a network entity and thus should not be processed.
7. Continues through the loop to get the next data record from DISCO.
8. Creates a `RIV::Record` object by calling the `RIV::Record` constructor. In this call, the data returned by `RIV::GetResult` to the *$data* variable is passed. This data is actually a reference to a hash list, which is the mechanism used to store network entity records from DISCO.

   The `RIV::Record` constructor returns the newly created `RIV::Record` object to the *$TestNE* variable.

   This code also handles a request from ncp_disco to terminate the Perl discovery agent if the test evaluates to true.
9. Assigns the value 1 to the `m_LastRecord` key in the hash.
10. Assigns the string `PerlDetails` to the `m_UpdAgent` key in the hash.
11. Assigns the value associated with the `m_IpAddress` key in the hash to the `$host` variable.

```
  print "Checking if IP address valid..\n"; 1
  if (!RIV::IsIpValid($host)){ 2
     print "Device has invalid IP address ignoring the record!\n";
     next DEVICE;
  }

  print "Checking if its a router...\n";
  $refNextHop=$agent->SnmpGet($TestNE,'ipForwarding');
```

```
 if ($refNextHop->{VALUE} != 1){  3
    print "Device is not a router!\n";
    next DEVICE;
 }

 print "Entering Mediation layer\n";
 Mediation();  4

 print "Entering Processing layer\n";
 Processing();  5

 print "Sending Record to Discovery\n";
 $agent->SendNEToDisco($TestNE,0);  6
}  7

sub Init{
 my $param=new RIV::Param();
 $agent=new RIV::Agent($param, "PerlDetails");
}  8
```

The following list explains specific numbered items in the previously listed IP routing discovery agent Perl program:

1. This section of code is the Mediation Filter. Check that the device has a valid IP address and also perform an SNMP get for `ipforwarding`. IP routers have a value of `'ipforwarding'` =1.
2. Check that the NE has a valid IP address.
3. Check if the NE is a router. It is a router if the *ipForwarding* value is 1.
4. This section of code is the Mediation Layer. Get the relevant SNMP information.
5. This section of code is the Processing Layer. Interpret the information to find the local and remote neighbors.
6. Send the record to DISCO.
7. The main loop ends.
8. Creates a new agent with the name `PerlDetails`.

```
sub Mediation{  1

 $refLifindex=$agent->SnmpGetNext($TestNE,'ipAdEntIfIndex');  2
 $refLnetmask=$agent->SnmpGetNext($TestNE,'ipAdEntNetMask');  2
 $refLphysaddress=$agent->SnmpGetNext($TestNE,'ifPhysAddress');  2

 $refRifindex=$agent->SnmpGetNext($TestNE,'ipRouteIfIndex');  3
 $refRtype=$agent->SnmpGetNext($TestNE,'ipRouteType');  3
 $refRnexthop=$agent->SnmpGetNext($TestNE,'ipRouteNextHop');  3

}

sub Processing{  4
 print "Processing the local neighbours\n";
 for ($j=0;$j<=$#$refLifindex;$j++){
   if (RIV::IsIpValid($refLifindex->[$j]->{ASN1}))
   {
      print $j, "\n";
      my %localNbr
      $localNbr{'m_IpAddress'} = $refLifindex->[$j]->{ASN1};  5
      $localNbr{'m_IfIndex'} = $refLifindex->[$j]->{VALUE};  5
      $localNbr{'m_NetMask'} = GetValueForKey($refLnetmask,  5
     $refLifindex->[$j]->{ASN1});

  $m_1 = $localNbr{'m_IpAddress'};
  $m_2 = $localNbr{'m_NetMask'};
  $localNbr{'m_SubNet'} = inet_ntoa(pack("N", (unpack ("N",
        inet_aton($m_1)) & unpack("N",
        inet_aton($m_2))) ));

  for ($i =0; $i <= $#$refLphysaddress; $i++)
  {
    if ($refLphysaddress->[$i]->{ASN1} == $refLifindex
        ->[$j]->{VALUE})
  {
   $localNbr{'m_LocalNbrPhysAddr'} = $refLphysaddress
         ->[$i]->{VALUE};
  }
 }
 print $localNbr{'m_IpAddress'}, $localNbr{'m_IfIndex'},
   $localNbr{'m_NetMask'}, "\n";
 $TestNE->AddLocalNeighbour(\%localNbr);
 }
}

print "processing for Remote Neighbours \n"; 6
@localN = $TestNE->GetLocalNeighbours();
for ($j=0;$j<=$#$refRtype;$j++)
{
 if(($refRtype->[$j]->{VALUE} !=2) && RIV::IsIpValid($refRtype
  ->[$j]->{ASN1}))
 {
   $nexthop = GetValueForKey($refRnexthop, $refRtype->[$j]->{ASN1});
   print "next hop = $nexthop for key $refRtype->[$j]->{ASN1}\n";
   if( NotLocalNbr($nexthop))
   {
     my %remoteNbr;
     $remoteNbr{'m_IpAddress'} = $nexthop;
     $Rindex = GetValueForKey($refRifindex,$refRtype->[$j]->{ASN1});
     AttachLocalNbr(\%remoteNbr, $Rindex);
       }
    }
 }
}
```

The following list explains specific numbered items in the previously listed IP routing discovery agent Perl program:

1. This section of code implements the Mediation Layer. You should perform all SNMP GET operations in this layer.
2. Get information required for determining local neighbors.
3. Get information required for determining remote neighbors.
4. This section of code implements the Processing Layer. To get the local neighbors, loop through `ipAdEntIfIndex` values. If the ASN1 value is a valid IP address, set the local neighbor tags for local neighbor IP address and `ifIndex`. Set tags for the `netMask` and `physaddress` based on the corresponding values in `ipAdEntIfIndex` and `ifAdEntNetMask`.

   To get the remote neighbors, start looping through the `ipRouteType`. The progam processes only if the route type is not 2 and the ASN1 value is a valid IP address. Then get the corresponding value of the next hop. Make it a remote neighbor if it is not a local neighbor IP address and the remote IP address does not already exist. Use the `ipRouteIfIndex` values to find the local neighbor to attach this remote neighbor to.
5. Add local neighbors to device record `m_IpAddress`, `m_IfIndex`, and `m_NetMask`.
6. This section of code determines and adds remote neighbors.

```
 sub GetValueForKey 1
 {
  my $refArray = shift;
  my $key = shift;
  for (my $jj=0; $jj<=$#$refArray; $jj++)
  {
    if ($refArray->[$jj]->{ASN1} eq $key)
    {
   return $refArray->[$jj]->{VALUE};
    }
  }
  return 0;
 }

 sub NotLocalNbr 2
 {
  my $ip_in = shift;
  @lN = $TestNE->GetLocalNeighbours();
  foreach $l_nbr (@lN)
  {
    print "RMAA $l_nbr->{m_IpAddress}, $ip_in, \n";
    if ($l_nbr->{'m_IpAddress'} eq $ip_in)
    {
   print "remote neighbour IP same as local neighbour IP \n";
   return 0;
    }
  my @remoteN = $TestNE->GetRemoteNeighbours($l_nbr);
  print @remoteN, "\n";
  foreach my $remoteNbr (@remoteN)
  {
     print "$remoteNbr->{'m_IpAddress'}, $ip_in, \n";
     if ($remoteNbr->{'m_IpAddress'} eq $ip_in)
     {
   print "remote neighbour IP already exists \n";
   return 0;
     }
   }
  }
  return 1;
 }
```

The following list explains specific numbered items in the previously listed IP routing discovery agent Perl program:

1. This section of code returns a value corresponding to the key from an array of `varOps`.
2. This section of code checks if the remote IP is the same as the local neighbor.

```
sub AttachLocalNbr 1
{
 my $refR = shift;
 my $index = shift;
 foreach $lnbr (@localN)
{
    if ($lnbr->{'m_IfIndex'} eq $index)
    {
  print $lnbr->{'m_IfIndex'}, $index, "\n";
  print "$lnbr->{'m_IpAddress'}connected to $refR->{m_IpAddress}\n";
  $TestNE->AddRemoteNeighbour($lnbr, $refR);
    }
  }
}
```

The following list explains specific numbered items in the previously listed IP routing discovery agent Perl program:

1. This section of code finds the appropriate local neighbor to connect the remote neighbor to.

Prototype agent definition file template

A discovery agent requires a discovery agent definition file (`$NCHOME/precision/disco/agents/*.agnt`), regardless of whether the agent is text-based or precompiled. Items that can be defined in this file include when and from where the discovery agent can be run; a list of devices that should be sent to the discovery agent; and the discovery phase at which the discovery agent should complete.

The following is a discovery agent definition file template with required and optional fields. Use this template to create the *.agnt file to be associated with your discovery agent. Explanations of specific lines follow the example.

Note: Where parsing errors occur, the discovery agent definition file rule will be ignored or the default behavior will be assumed.

See the *IBM Tivoli Network Manager IP Edition Discovery Guide* for more information on the discovery agent definition file and its associated keywords.

```
--
    -- The following fields are used to initialize the config GUI
    -- and update DiscoAgents.cfg when the agent is first installed
    -- The DiscoAgentDescription keyword provides a way to describe
    -- the discovery agent. The CustomPerlAgent is used as an example.
    DiscoAgentDescription("Agent description goes here.");

    DiscoAgentGUILocked( 0 );
    DiscoAgentClass( 0 );
    DiscoAgentIsIndirect( 0 );
    DiscoAgentPrecedence( 2 );
    DiscoAgentEnabledByDefaultOnPartial( 0 );
    DiscoAgentEnabledByDefault( 0 );
    DiscoAgentDefaultThreads( 1 );

-- Discovery agent type section
DiscoCompiledAgent
{
--
-- Optional "ncp_ctrl" information section
--
-- DiscoExecuteAgentOn("<Machine Name>");
--
-- Devices that should be sent to this agent section
--
DiscoAgentSupportedDevices
(
"Filter Expression"
);
-- Agent completion phase section
--
DiscoAgentCompletionPhase( n );

-- Mediation filter section
--

DiscoAgentMediationFilter
{
// Optional section containing filters for the mediation layer.
}
}
```

The following list explains specific items in the Agent Definition File template:

- **Discovery agent type section**

Specifies the agent type. The template specifies the keyword `DiscoCompiledAgent` that denotes a compiled discovery agent. This compiled agent has a corresponding shared library in the `$NCHOME/precision/lib` directory. Possible other values include `DiscoDefinedAgent` and `DiscoCombinedAgent`.

This section is required.

- **Optional "ncp_ctrl" information section**

  Specifies **ncp_ctrol** information. This control information defines when and from what server or computer the discovery agent is to be run. If this line is omitted, the discovery agent will be run on the same server or computer as the DISCO process. Replace *Machine Name* with the name of the server or computer on which the discovery agent is to be run.

  This section is optional.

- **Devices that should be sent to this agent section**

  Specifies the devices that should be sent to the discovery agent. Replace *"Filter Expression"* with valid values that include a range or list of IP addresses to include or exclude, along with a range or list of OIDs to include or exclude. The default is ALL.

  This section is required.

  The following is an example:

```
DiscoAgentSupportedDevices
 (
  "(
   m_ObjectId LIKE '1\.3\.6\.1\.4\.1\.9\.5\.'
   )
   OR
   (
   (
    m_ObjectId LIKE '1\.3\.6\.1\.4\.1\.9\.1\.'
   )
   AND
   (
    m_Description NOT LIKE
     'IOS \(tm\) C2900XL Software \(C2900XL-C3H2S-M\),
          Version 12.0\(5.3\)WC\(1\), MAINTENANCE INTERIM SOFTWARE'
    AND
     m_ObjectId <> '1.3.6.1.4.1.9.1.576'
    AND
     m_ObjectId <> '1.3.6.1.4.1.9.1.577'
    AND
     m_ObjectId <> '1.3.6.1.4.1.9.1.577'
    AND
     m_ObjectId <> '1.3.6.1.4.1.9.1.619'
   )
  )"
 );
```

- **Agent completion phase section**

  Specifies at which of the *n* discovery phases should this discovery agent complete.

  This section is required.

  The following example shows that the discovery agent should complete at phase 3:

```
--
 -- During which of the n discovery phases should this agent complete?
 --
 DiscoAgentCompletionPhase( 3 );
```

- **Mediation filter section**

Specifies the mediation layer, which contains, among other items, an optional filter on the mediation layer.

Using threads in discovery agents

Discovery agents written in Perl can experience slow performance because the data retrieved for each device operating in the network is retrieved within a single thread. Thus, the discovery agent spends much of its time idle, waiting for data to be retrieved from a device. The `RIV::Agent` module provides a multithreading capability that improves the performance of discovery agents.

The following topics provide information about the `RIV::Agent` module multithreading capability.

Note: Although Perl itself supports execution of multiple threads, many of the add-on CPAN modules often used with Perl are not thread safe. This means that Perl discovery agents using CPAN modules may need to be restricted to a single thread.

The `RIV::Agent::LockThreads` and `RIV::Agent::UnLockThreads` methods might enable you to use third party Perl modules. These methods allow you to restrict access to a section of a discovery agent to a single thread.

Discovery agent threads example

To overcome the slow performance of discovery agents written in Perl, the `RIV::Agent` module provides a multithreading capability. This capability allows you to serialize execution of specific sections of a discovery agent.

The following example calls the `LockThreads` method to serialize execution of specific sections of a discovery agent:

```
#
    # Begin a serialised section of execution within the Perl agent
    #
    $agent->LockThreads();

    #
    # It is important not to have any fatal errors that could prevent the
    # threads getting unlocked again so wrap the following in an eval block.
    #
    eval
    {
           ... only one agent thread executing here at any given time ..
    }
    if ($@)
    {
        warn "Error: $@\nWhen executing serialised block\n";
    }
 #
    # Unlock to allow other threads a chance to execute the above section
    #
    $agent->UnLockThreads();
```

Note: It may be possible to use a non-thread safe Perl module within such a serialized section of the discovery agent. However, no guarantees can be made and results may vary with different modules. So if the results are not successful then the discovery agent may still need to be restricted to a single thread.

Default number of threads

The default number of threads for a Perl discovery agent is defined within its agent definition file.

The number of threads that a Perl discovery agent should use is defined in exactly the same way as that for normal discovery agents, by modifying the `m_NumThreads` attribute in the `DiscoAgents.cfg` configuration file. The default number of threads for a Perl discovery agent is specified in the agent's definition file as follows:

```
DiscoAgentDefaultThreads( 10 );
```

Note: In order for a Perl discovery agent to use multiple threads, a `DiscoAgentDefaultThreads` entry must be defined in its agent definition file. If such an entry does not exist, then the value of the `m_NumThreads` attribute in the `DiscoAgents.cfg` configuration file will be ignored.

Chapter 3. Accessing component databases

Network Manager IP Edition provides component databases that store specific categories of information. Each component schema can consist of one or more databases and each database one or more tables. For example, the ncp_class database consists of one database and three tables. You use the `RIV::OQL` Perl API module to access or modify records in these component databases.

To access or modify records in the Network Manager IP Edition component databases, you use the `RIV::OQL` Perl API module. Typically, you will also make use of the `RIV::App` and `RIV::Param` Perl API modules.

See the *IBM Tivoli Network Manager IP Edition Management Database Reference* (SC27-2767-00) for information about the component databases you can access with the `RIV::OQL` module.

Object Query Language

Object Query Language (OQL) is a version of the Structured Query Language (SQL) that has been designed for use in Network Manager IP Edition. The components create and interact with their databases using OQL.

Use OQL to create new databases or insert data into existing databases (to configure the operation of Network Manager IP Edition components) by amending the component schema files. You can also issue OQL statements using the OQL Service Provider, for example, to create or modify databases, insert data into databases and retrieve data.

For more information about the OQL schema used by Network Manager IP Edition, see the *IBM Tivoli Network Manager IP Edition Management Database Reference*.

The OQL Service Provider is described in the *IBM Tivoli Network Manager IP Edition Administration Guide*.

Differences between OQL and Structured Query Language

Network Manager IP Edition uses OQL to transfer data between and communicate with its internal databases. OQL is an object-based version of Structured Query Language (SQL) that was designed specifically around the operational needs of the Network Manager IP Edition architecture.

The following items identify the main differences between OQL and SQL:
- OQL has the ability to support object referencing within database tables. Thus, it is possible to have objects nested within objects.
- Not all SQL keywords are supported within OQL. Thus, irrelevant keywords have been removed for the OQL syntax.

Note: Before using the Perl API to access the component databases, make sure you understand the available component databases and the Object Query Language. See *IBM Tivoli Network Manager IP Edition Management Database Reference* (SC27-2767-00) for details related to the component databases. See *IBM Tivoli*

Network Manager IP Edition Language Reference (SC27-2768-00) for details related to the Object Query Language.

Actions that can be performed on component databases

The Perl API provides utilities that allow you to query any of the Network Manager IP Edition component databases and to retrieve and control the information stored in them.

The `RIV::OQL` Perl API module allows you to perform operations on the component databases, including:
- Inserting new entries into the component databases
- Modifying existing device attributes
- Deleting entries from the component databases

To access a particular component database, the service in which the database resides must be running. For example, if you want to access a component database that resides in DISCO, **ncp_disco** must be running. Likewise, to access a CLASS database, **ncp_class** must be running. For a full listing of services to which you can connect, see "RIV::OQL Constructor" on page 97.

After you create a `RIV::OQL` object based on a selected service, you can perform one of four actions on a component database:
- SELECT
- INSERT
- UPDATE
- DELETE

A SELECT statement returns records, while the other statements do not return any records.

The `RIV::OQL` module allows you to:
- Access retrieved records using the `RIV::GetResult` method.
- Print these retrieved records using the `RIV::OQL::Print` method.
- Create new databases and tables with the `RIV::OQL::CreateDB` and `RIV::OQL::CreateTable` methods.

In addition to the previously described convenience methods, you can use the `RIV::OQL::Send($statement, $returnResults)` method, where `$statement` is any valid OQL statement, and `$returnResults` equals:
- 1 — For queries that return results. For example, select and show.
- 0 — For queries that do not return results. For example, insert.

Example Perl scripts that operate on component databases

Use the following examples as guides to writing Perl scripts that access the Network Manager IP Edition component databases.

The oql_example.pl example script

The **oql_example.pl** script demonstrates the use of Perl to parse the `/etc/hosts` file and input any devices listed therein into the **finders.despatch** database, providing the IP address listed is valid.

The **oql_example.pl** script uses some of the methods provided in the `RIV::Param` and `RIV::OQL` Perl API modules. The example is divided into two sections: lines 1-15 and lines 16-33. Explanations follow lines 1-15 and lines 16-33.

```
#!/opt/netcool/precision/Solaris2/bin/ncp_perl
1 use strict;
2 use RIV;
3 use RIV::Param;
4 use RIV::App;
5 use RIV::OQL;
6
7 my $param = new RIV::Param() or
8 die "RIV::Param::new failed";
9
10 my $app = new RIV::App($param, "ncp_test:oql") or
11 die "Can't create RIV application session";
12
13 my $oql = new RIV::OQL($app, "Disco") or
14 die "Can't create RIV OQL session";
15
```

The following list explains specific numbered items in the previously listed Perl script example:

- **Lines 1-5**

  Declare the Perl API modules to use in the **oql_example.pl** script. This script makes use of the `RIV`, `RIV::Param`, `RIV::App`, and `RIV::OQL` modules. The **oql_example.pl** script also makes use of the `use strict` pragma. The `use strict` pragma enforces good programming practices, including enforcing the declarations of any new variables with `my`.
- **Lines 7-8**

  Creates and initializes a new `Param` object. If the `Param` object cannot be created the script stops. In this call to the `RIV::Param` constructor, no arguments are specified. This means that there are no application-specific command line arguments. However, the standard command line arguments that the `RIV::Param` module provides are available once the `Param` object is created.
- **Lines 10-11**

  Creates a new client/server application object using the `RIV::App` constructor. This call to the `RIV::App` constructor takes two parameters:
  - `RIV::Param` — Specifies a `RIV::Param` object reference. This object was returned to `$param` by the `RIV::Param` constructor.
  - `$progname` — Specifies a string that uniquely identifies this application. By convention, the application name should start with `ncp_`. In the example, the specified application name is `ncp_test:oql`.

    If the client/server application object cannot be created, the script stops.
- **Lines 13-14**

Creates a new OQL object on the service type Disco. If the OQL object fails to create, the script stops.

```
16 open (INPUT, "/etc/hosts")
17 or die "Could not open /etc/hosts: $!\n":
18 my $number_records =0;
19 my $finder = "PerlFileFinder";
20 my $sep = "'";
21 my $stat;
22 while (<INPUT>){
23
24  next if (/^#/ or /^--/);
25  my ($ipadd, $ipname)= /^(\S+)\s+(\S+)/;
26
27  if (RIV::IsIpValid($ipadd)){
28     my %record = (m_Creator => $sep.$finder.$sep, m_Name =>
       $sep.$ipname.$sep, m_IpAddress => $sep.$ipadd.$sep, );
29  $oql->Insert('finders', 'despatch', \%record);
30  $number_records ++;
31  }
32 }
33 print STDOUT "Number of Records input = ", $number_records;
```

- **Line 16**

  Opens and reads the file `/etc/hosts` using the name INPUT as a file handle.
- **Lines 18-21**

  Initializes variables.
- **Line 24**

  Ignores lines beginning with # or -- characters.
- **Line 25**

  Browse each line and get the IP address and name, filtering out any spaces in between.
- **Line 27**

  Checks the validity of the IP address.
- **Line 28**

  If the IP address is valid, the value pairs `m_Creator`, `m_Name`, and `m_IpAddress` are put in the *%record* hash.
- **Line 29**

  Inserts the record in the despatch table of the finders database.
- **Line 33**

  After the loop completes, prints to standard output the number of records input into the finders.despatch table. The finders database is defined in the DiscoSchema.cfg file.

  See the *IBM Tivoli Network Manager IP Edition Discovery Guide* (SC27-2762-00) for information about the finders database.

OQL example script

The OQL example script shows how to create an OQL session and perform several operations on the MODEL database.

The example is divided into two sections: lines 1-26 and lines 27-48. Explanations follow lines 1-26 and lines 27-48.

```
# $PRECISION_HOME/bin/ncp_perl
1 use RIV;
2 use RIV::Param;
3 use RIV::App;
4 use RIV::OQL;
5
6 my $param = new RIV::Param()
7 or die "RIV::Param::new failed";
8
9 my $app = new RIV::App($param, "ncp_test:oql");
10 or die "Can't create RIV application session";
11
12 my $oql = new RIV::OQL($app, "Model");
13 or die "Can't create RIV OQL session";
14
15 my $stat ='insert into master.entityByName (EntityName, Description,
ClassName) values ("bar", "This is a switch", "Switch");';
16 $oql->Send($stat, 0);
17
18 $stat = 'select * from master.entityByName;';
19 $oql->Send($stat);
20 my ($type, $data) = $oql->RIV::GetResult(10);
21 $oql->Print($data);
22
23 $oql->Select('master', 'entityByName', 'ALL');
24 ($type, $data) = $oql->RIV::GetResult(10);
25 $oql->Print($data);
26
```

The following list explains specific numbered items in the previously listed Perl script example:

- **Lines 1-4**

  Declare the Perl API modules to use in the OQL example script. This script makes use of the `RIV`, `RIV::Param`, `RIV::App`, and `RIV::OQL` modules.
- **Line 6**

  Reads and parses the command line. The standard arguments are hidden.
- **Lines 9-10**

  Creates a new `RIV` application object. If the creation of this object fails, the script stops.
- **Lines 12-13**

  Creates an `OQL` object with service `Model`. If the creation of this object fails, the script stops.
- **Lines 15-16**

  Inserts a record into MODEL using the `Send` method.
- **Lines 18-19**

  Determines what records there are in the MODEL database using the `Send` method.
- **Lines 20-21**

  Gets and prints the records.
- **Lines 23-25**

Determines what records there are in the MODEL database using the `Select` method.

```
27 $oql->Delete(master, entityByName, "ClassName = 'Switch'");
28
29 my %insert_rec;
30 $insert_rec{EntityName} = "'foo'";
31 $insert_rec{Description} = "'This is a router'";
32
33 $oql->Insert('master', 'entityByName', \%insert_rec);
34
35 $oql->Select('master', 'entityByName', 'ALL');
36 ($type, $data) = $oql->RIV::GetResult(10);
37 $oql->Print($data);
38
39 $oql->CreateDB("PerlDB");
40
41 my %table_columns = (m_IpAddress=> "text", m_Name=> "text");
42 $oql->CreateTable("PerlDB", "PerlTable", \%table_columns);
43 my %dummy_entry = ("m_IpAddress"=> "'8.9.10.11'", "m_Name" => "'dummy'");
44 $oql->Insert('PerlDB', 'PerlTable', \%dummy_entry);
45
46 $oql->Select('PerlDB', 'PerlTable', 'm_Name');
47 ($type, $data) = $oql->RIV::GetResult(10);
48 $oql->Print($data);
```

The following list explains specific numbered items in the previously listed Perl script example:

- **Line 27**

  Deletes the record using the `Delete` method.
- **Lines 29-34**

  Inserts a new record.
- **Lines 35-37**

  Checks if the records were deleted or inserted.
- **Line 39**

  Creates a database called `PerlDB`.
- **Lines 41-42**

  Creates a table in the `PerlDB` with columns `m_IpAddress` and `m_Name`.
- **Lines 43-44**

  Inserts a record into the `PerlDB` database.
- **Lines 46-48**

  Selects the entries in the user defined database to verify that the database table has been created and the record inserted.

Chapter 4. Performing SNMP queries

The RIV::SnmpAccess module allows client/server scripts that use the Perl API to retrieve SNMP information from a network device through the SNMP helper.

To write client/server Perl scripts that retrieve SNMP information from network devices you typically use the RIV::Param, RIV::App, and RIV::SnmpAccess modules.

Note: Discovery agent Perl scripts should not use the RIV::SnmpAccess module. Discovery agent Perl scripts use the RIV::Agent module, which provides its own methods to perform SNMP operations through the SNMP helper.

Using get methods to obtain SNMP information from a device

The Perl API, specifically the RIV::SnmpAccess module, provides a number of get methods for retrieving SNMP information from a particular device. You can make these get SNMP information requests either synchronously or asynchronously because the RIV::SnmpAccess module provides both synchronous and asynchronous versions of these get methods.

The following table summarizes which methods to call in a client/server Perl script.

Synchronous/asynchronous method	Description	Level of SNMP information accessed
SnmpGet and AsyncSnmpGet	The caller specifies a valid IP address for the particular device and the MIB variable of interest. These methods return the specified MIB variable for the specified device.	These methods retrieve a single MIB variable. If you pass a MIB table (instead of a single MIB variable) to these methods, only the first entry in this MIB table is returned.
SnmpGetNext and AsyncSnmpGetNext	The caller specifies a valid IP address for the particular device and a MIB table of interest. These methods return the specified MIB table for the specified device.	These methods retrieve an entire MIB table that contains multiple variables (for example, ifTable).
SnmpGetBulk and AsyncSnmpGetBulk	The caller specifies a valid IP address for the particular device and the MIB variables of interest. These methods return the specified MIB variables for the specified device.	These methods retrieve multiple MIB variables (for example, ifDescr, ifType, and ifSpeed).

Making synchronous and asynchronous SNMP get requests

The Perl API provides two ways to make SNMP get requests: synchronous and asynchronous. You make these SNMP get requests by calling the get request methods that the `RIV::SnmpAccess` module provides.

The following list briefly describes the two ways to perform SNMP get requests:

- Synchronous — Each successive transmission of data requires a response to the previous transmission before a new one can be initiated.
- Asynchronous — Each transmission of data proceeds independently until one transmission needs to interrupt another one with a request.

When writing a client/server Perl script that makes a synchronous SNMP GET request, the request is made and the caller will not be able to perform other tasks until the specified MIB variable has been retrieved. The information that is returned will have an attached tag so that you know what it is referring to. The tag will be whatever you have specified it to be in the synchronous get method.

Unlike a synchronous SNMP GET request, an asynchronous SNMP GET request is multithreaded. Thus, you are free to perform other tasks while waiting for a response when using an asynchronous SNMP GET request. In the Perl API, the caller can specify the number of threads. When retrieving SNMP information from a large device, use 10 threads.

Example SNMP GET access script

The SNMP GET access example Perl script shows how to retrieve SNMP information using several of the SNMP get methods — both synchronous and asynchronous — that the `RIV::SnmpAccess` module provides. Use this example as a model for writing your own client/server Perl scripts that retrieve SNMP information for specific devices from a single MIB variable, multiple MIB variables, and a MIB table.

Declare Perl API modules and variables

This section of the SNMP GET access example Perl script declares the Perl API modules to be used as well as a number of variables. Use this part of the example script as a guide to setting up client/server Perl scripts that will retrieve SNMP information.

The SNMP GET access example Perl script declares the Perl API modules to be used and a number of variables as follows:

```
#!$NCHOME/bin/ncp_perl
use strict;
use RIV;
use RIV::Param;
use RIV::App;
use RIV::SnmpAccess; # qw (RivSnmpResultOk);

$RIV::SnmpAccess::MaxAsyncConcurrent = 40;

my $Ttype = "TEST";

my $Verbose;
my @_Usage = ("node" [async]);

```

The following list explains specific numbered items in the previously listed section of the SNMP GET access Perl script example.

- **Line 1**

  Declares the `strict` pragma with the `use` directive. The `strict` pragma enforces good programming practices, including enforcing the declarations of any new variables with `my`.
- **Lines 2-5**

  Specify the `use` directive to declare the Perl API modules to be used. In this case, use the `RIV`, `RIV::Param`, `RIV::App`, and `RIV::SnmpAccess` modules.
- **Line 7**

  Sets the *MaxAsyncConcurrent* `RIV::SnmpAccess` module variable to the value 40. This module variable sets the maximum number of concurrent asynchronous SNMP get requests.
- **Lines 9-12**

  Declare the following `my` variables:
  - *$Ttype* — Stores a string that identifies whether the SNMP GET access is synchronous or asynchronous. Later sections of the SNMP GET access example script use this variable in calls to the `print` operator. The variable gets set initially to the string `TEST`.
  - *$Verbose* — Specifies how the script progress details are to be displayed. The `-v` option displays verbose progress details. This variable is defined with the `RIV::Param` module, specifically with the `Usage` method.
  - *@_Usage* — Specifies the usage string suffixes `node` and `async`.

Create and initialize a RIV::Param object

This section of the SNMP GET access example Perl script creates and initializes a new `RIV::Param` object. Use this part of the example script as a guide to creating and initializing new `RIV::Param` objects in client/server Perl scripts that retrieve SNMP information.

The SNMP GET access example Perl script creates and initializes a new `RIV::Param` object as follows:

```
14 my $param = new RIV::Param({"-v"=> [ $RIV::Param::NoArg, \$Verbose ],},
    \@_Usage) or
15 die "RIV::Param::new failed";
16
17 my $node = shift @ARGV;
18 my $what = shift @ARGV;
19 $what = "" unless defined $what;
20
21 $param->Usage(1) unless (defined $node && $node ne "");
```

The following list explains specific numbered items in the previously listed section of the SNMP GET access Perl script example:

- **Line 14**

  Creates and initializes a new `RIV::Param` object by calling the `RIV::Param` constructor.

  Note: The standard arguments (for example, `-domain`, `-debug`, and so forth) are hidden.
- **Line 15**

If the constructor fails to create and initialize the new `RIV::Param` object, consider this a fatal error and call the `die` function. The `die` function prints out an appropriate message (in this case, that the `RIV::Param` constructor failed) to the standard error stream.

Create and initialize a RIV::App object

This section of the SNMP GET access example Perl script creates and initializes a new `RIV::App` object. Use this part of the example script as a guide to creating and initializing new `RIV::App` objects in client/server Perl scripts that retrieve SNMP information.

The SNMP GET access example Perl script creates and initializes a new `RIV::App` object as follows:

```
22 my $app = new RIV::App($param, "ncp_test:snmp") or

23 die "Can't create RIV application session" unless (defined $app);

24
```

The following list explains specific numbered items in the previously listed section of the SNMP GET access Perl script example:

- **Line 22**

  Creates and initializes a new `RIV::App` object by calling the `RIV::App` constructor. This call to the `RIV::App` constructor takes the following parameters:
  - `RIV::Param` — Specifies a reference to a `RIV::Param` object. In this example, the newly created `RIV::Param` object is returned to the `my $param` variable in a previous call to the `RIV::Param` constructor.
  - `$progname` — Specifies a string that uniquely identifies an application. In this example, the application name is identified by the string `ncp_test:snmp`.
- **Line 23**

  If the constructor fails to create and initialize the new `RIV::App` object, consider this a fatal error and call the `die` function. The `die` function prints out an appropriate message (in this case, that the `RIV::App` constructor failed) to the standard error stream.

Create and initialize RIV::SnmpAccess object

This section of the SNMP GET access example Perl script creates and initializes a new `RIV::SnmpAccess` object. Use this part of the example script as a guide to creating and initializing new `RIV::SnmpAccess` objects in client/server Perl scripts that retrieve SNMP information.

The SNMP GET access example Perl script creates and initializes a new `RIV::SnmpAccess` object as follows:

```
25 my $snmp = new RIV::SnmpAccess($app) or

26 die "Can't create RIV SNMP session";

27
```

The following list explains specific numbered items in the previously listed section of the SNMP GET access Perl script example:

- **Line 25**

Creates and initializes a new `RIV::SnmpAccess` object by calling the `RIV::SnmpAccess` constructor. Upon successful completion, the `RIV::SnmpAccess` constructor returns a `RIV::SnmpAccess` object to the `my $snmp` variable.

This call to the `RIV::SnmpAccess` constructor takes the following parameter:

– `$rivSession` — Specifies a reference to `RIV::App` object returned in a previous call to the `RIV::App` constructor. In this example, the newly created `RIV::App` object is returned to the `my $app` variable in a previous call to the `RIV::App` constructor.

- **Line 26**

  If the constructor fails to create and initialize the new `RIV::SnmpAccess` object, consider this a fatal error and call the `die` function. The `die` function prints out an appropriate message (in this case, that the `RIV::Snmp` constructor failed) to the standard error stream.

Check the device IP address and node name

This section of the SNMP GET access example Perl script checks for the device's IP address and node name. Use this part of the example script as a guide to writing code that checks for a device's IP address and node name in client/server Perl scripts that retrieve SNMP information.

The SNMP GET access example Perl script checks for a device's IP address and node name as follows:

```
28 my $nodeIP = $node;

29 if ($node !~ /^\d+\.\d+\.\d+\.\d+$/) {

30 $nodeIP = gethostbyname($node);

31 $nodeIP = inet_ntoa($nodeIP) if (defined $nodeIP) or

32 die "Can't find IP address for '$node'";

33 }

34
```

The following list explains specific numbered items in the previously listed section of the SNMP GET access Perl script example:

- **Line 28**

  Assigns the value stored in the `my $node` variable to the `my $nodeIP` variable. The `my $node` variable was set after the call to the `RIV::Param` constructor.

  See "RIV::Param Constructor" on page 106 for more information.

- **Line 29**

  Determines if an IP address or node (host) name was specified.

- **Lines 30-31**

  Gets the IP address from the node name by calling the `gethostbyname` and `inet_ntoa` functions.

- **Line 32**

  If the `defined` function verifies that the value in `$nodeIP` is `undef`, consider this a fatal error and call the `die` function. The `die` function prints out an appropriate message (in this case, that the IP address for this device cannot be found) to the standard error stream.

Determine which SNMP GET requests to run

This section of the SNMP GET access example Perl script determines which SNMP GET requests — synchronous or asynchronous — to run. Use this part of the example script as a guide to writing code that sets up an appropriate condition to run either the synchronous or asynchronous SNMP GET requests in client/server Perl scripts that retrieve SNMP information.

The SNMP GET access example Perl script sets up a condition to run either the synchronous or asynchronous SNMP GET requests as follows:

```
35 if ($what =~ /async/) {

36 $Ttype = "ASYNCTEST";

37 AsyncTests();

38 exit 0;

39 }
40
41 SyncTests();
42
43 exit 0;
44
45 ##########################################################################
```

The following list explains specific numbered items in the previously listed section of the SNMP GET access Perl script example:

- **Lines 35-39**

  Checks the input parameter to determine if the asynchronous tests (using the asyncrhonous SNMP GET requests) should be run.
- **Line 41**

  By default, the synchronous tests (using the synchronous SNMP GET requests) should be run.

Perform asynchronous SNMP GET requests

This section of the SNMP GET access example Perl script sets up the logic to run the asynchronous SNMP GET requests. Use this part of the example script as a guide to writing code that makes use of some of the asynchronous SNMP GET requests in client/server Perl scripts that retrieve SNMP information.

The SNMP GET access example Perl script sets up the logic and runs the asynchronous SNMP GET requests as follows:

```
46 sub AsyncTests {
47
48 my $sTag = "GETNEXT_$node";
49 print "$Ttype: call AsyncSnmpGetNext($sTag, $nodeIP, ". "NULL,
 ifDescr)\n" or
50 die "AsyncSnmpGetNext() failed";
51
52 my ($tag, $data) = $snmp->GetResult(-1) or
53 die "Unexpected tag '$tag'";
54

55 foreach my $varop (@{ $data->[0] })
56 {
57 PrintVarOp($varop);
58 }
59
60 $sTag = "GET_$node";
61 print "$Ttype: call AsyncSnmpGet($sTag, $nodeIP, NULL, "ifDescr" 2)\n"
or
62 die "AsyncSnmpGet() failed" ;
63
64 ($tag, $data) = $app->GetResult(-1) or
65 die "Unexpected tag '$tag'" unless ($tag eq "SNMP_$sTag");
66
67 PrintVarOp($data->[0]);
68
69 }
70
71 ########################################################################
```

The following list explains specific numbered items in the previously listed section of the SNMP GET access Perl script example:

- **Lines 48-50**

  Performs an SNMP GETNEXT asynchronous request (by calling the `AsynchSnmpGetNext` method).
- **Lines 52-58**

  Receives the results, using the `$snmp->RIV::GetResult()` method, and then prints these results.
- **Lines 60-69**

  Performs an SNMP GET asynchronous request (by calling the `AsynchSnmpGet` method). Receives the results using the `$snmp->RIV::GetResult()` method. Then checks the tag and prints the results.

Perform synchronous SNMP GET requests

This section of the SNMP GET access example Perl script sets up the logic to run the synchronous SNMP GET requests. Use this part of the example script as a guide to writing code that makes use of some of the synchronous SNMP GET requests in client/server Perl scripts that retrieve SNMP information.

The SNMP GET access example Perl script sets up the logic and runs the synchronous SNMP GET requests as follows:

```
72 sub SyncTests {
73
74 print "$Ttype: call SnmpGetNext($nodeIP, NULL, ifDescr)\n";
75 my ($vap) = $snmp->SnmpGetNext($nodeIP, "", "ifDescr") or
76 die "SnmpGetNext on ifDescr table for '$node' failed";
77
78 foreach my $varop (@{ $vap })
79 {
80  PrintVarOp($varop);
81 }
82
83 print "$Ttype: call SnmpGet($nodeIP, NULL, ifDescr,1)\n";
84 $vap = $snmp->SnmpGet($nodeIP, "", "ifDescr",1) or
85 die "SnmpGetNext on ifDescr table for '$node' failed";
86
87 PrintVarOp($vap);
88
89 print "$Ttype: call SnmpGetBulk($nodeIP, NULL,\@oids,8,100)\n";
90 my @oids=('sysDescr', 'sysContact', 'sysUpTime', 'ipInReceives',
  'ipOutRequests', 'ipOutDiscards', 'ipForwDatagrams',
  'tcpCurrEstab', 'ifDescr');
91
92 ($vap) = $snmp->SnmpGetBulk($nodeIP, "", \@oids, 8, 100);
93 foreach my $varop (@{ $vap })
94 {
95  PrintVarOp($varop);
96 }
97 }
98 ########################################################################
```

The following list explains specific numbered items in the previously listed section of the SNMP GET access Perl script example:

- **Lines 74-81**

  Performs an SNMP GETNEXT synchronous request (by calling the `SnmpGetNext` method).
- **Lines 83-87**

  Performs an SNMP GET synchronous request (by calling the `SnmpGet` method). Receives the results, using the `$snmp->RIV::GetResult()` method, and then prints these results.
- **Lines 91-97**

  Performs an SNMP GETBULK synchronous request (by calling the `SnmpGetBulk` method).

Print the SNMP varops

This section of the SNMP GET access example Perl script sets up the logic to print the SNMP varops. Use this part of the example script as a guide to writing code that prints the SNMP varops in client/server Perl scripts that retrieve SNMP information.

The SNMP GET access example Perl script sets up the logic to print SNMP varops as follows:

```
99 sub PrintVarOp {
100  my ($vp) = @_;
101
102 my $asn1 = $vp->{ASN1};
103  my $value = $vp->{VALUE};
104  my ($oid, $index, $name) = $snmp-> SplitOidAndIndex ($asn1);
105  print "$Ttype: $name.$index = $value\n";
106 }
```

The following list explains specific numbered items in the previously listed section of the SNMP GET access Perl script example:

- **Lines 99-106**

  Prints the SNMP varops.

Chapter 5. Writing and integrating Perl applications with third-party products

The Perl API allows you to write Perl applications (for example, Listeners) that you can then integrate with third-party products.

Listener applications

A Listener is an application written for a specific Network Manager IP Edition database. The purpose of a Listener is to "listen" and respond to record events that occur in the associated database.

Record events in the database include updates to existing records, additions of new records, and deletions of existing records. A Listener application can process these record events in order to:

- Update an external database
- Send email to an appropriate administrator or end user based on the event type
- Integrate with a variety of third-party products or applications

Users of the Perl API can also make use of the mail modules (for example, `Mail::Mailer`) to email database record events. Listener applications, through the `RIV::OQL` module, can send a stream of data into HTML, CGI scripts, and XML data.

Note: Communication with external databases — such as, Oracle® or Sybase® — can be done using the Perl DBI module.

In the record received from the Listener there is a tag for `Action Type` that defines the action performed. For example, a record returned with an action type of 2 indicates that the listener had picked up a record deletion. The actions are summarized in the table below.

Table 1. Listener actions

Tag	Action
0	Insert
1	Update
2	Delete

Note: The listener must be associated with a subject. For example, to listen to events the subject must be `ITNM/EVENT/NOTIFY`.

Example Listener script

The Listener example script shows how to "listen" to record insertions, deletions, and updates in the MODEL topology database. Use this example as a model for writing your own Listener applications using the Perl API.

Declare Perl API modules and variables for Listener

This section of the Listener example script declares the Perl API modules to be used. Use this part of the example script as a guide to declaring the Perl modules used with Listener applications.

The Listener example script declares the Perl API modules to be used as follows:

```
1 use RIV;
2 use RIV::Param;
3 use RIV::App;
4
```

The following list explains specific numbered items in the previously listed section of the Listener Perl script example:

- **Lines 2-5**

  Declare the Perl API modules to be used with the `use` directive, specifically, `RIV`, `RIV::Param`, and `RIV::App`.

Create and initialize a RIV::Param object for Listener

This section of the Listener example script creates and initializes a new `RIV::Param` object. Use this part of the example script as a guide to creating and initializing new `RIV::Param` objects in Listener applications.

The Listener example script creates and initializes a new `RIV::Param` object as follows:

```
5 $param = RIV::Param::new();
```

The following list explains the specific numbered item in the previously listed section of the Listener Perl script example:

- **Line 5**

  Creates and initializes a new `RIV::Param` object by calling the `RIV::Param` constructor. Upon successful completion, the `RIV::Param` constructor returns a `RIV::Param` object to the `$param` variable. This `RIV::Param` object is then passed as a parameter to the `RIV::App` constructor.

Create and initialize a RIV::App object for Listener

This section of the Listener example script creates and initializes a new `RIV::App` object. Use this part of the example script as a guide to creating and initializing new `RIV::App` objects in Listener Perl scripts.

The Listener example script creates and initializes a new `RIV::App` object as follows:

```
6 $app = RIV::App::new($param, "model_listener");
7
```

The following list explains the specific numbered item in the previously listed section of the Listener Perl script example:

- **Line 6**

Creates and initializes a new `RIV::App` object by calling the `RIV::App` constructor. This call to the `RIV::App` constructor takes the following parameters:

- `RIV::Param` — Specifies a reference to a `RIV::Param` object. In this example, the newly created `RIV::Param` object is returned to the `$param` variable in a previous call to the `RIV::Param` constructor.
- `$progname` — Specifies a string that uniquely identifies an application. In this example, the application name is identified by the string `model_listener`.

Bind the RIV::App object to the message broker subject for Listener

Network Manager IP Edition uses the message broker publish and subscribe messaging system to enable processes to communicate with each other. This section of the Listener example script binds the newly created `RIV::App` object to message broker. Use this part of the example script as a guide to binding `RIV::App` objects to message broker.

The Listener example script binds the newly created `RIV::App` object to message broker as follows.

See the *IBM Tivoli Network Manager IP Edition Administration Guide* (SC27-2761-00) for more information on message broker.

```
8 $ok = $app->RIV::AddSubject('ITNM/MODEL/NOTIFY','model');
9 print $ok, "\n";
```

The following list explains the specific numbered items in the previously listed section of the Listener Perl script example:

- **Line 8**

  Calls the `AddSubject` virtual method to bind the `RIV::App` object to message broker. The `AddSubject` virtual method takes two parameters:
  - `$subject` — Specifies the message broker subject to which the `RIV::App` object binds. In this call, the message broker subject is `ITNM/MODEL/NOTIFY`.

    Note: If you wanted the Listener application to listen to events, then `$subject` would be `ITNM/EVENT/NOTIFY`.
  - `$tag` — Specifies the tag to be appended to `USER_`, which describes the message returned through the `RIV::GetResult` method. In this call, the tag is `model`.

  Upon successful completion, the `AddSubject` virtual method returns the value 1.
- Calls the `print` operator to print the value that the `AddSubject` virtual method returns to the standard output.

Write database records to a log file

This section of the Listener example script sets up an appropriate loop for "listening" to records inserted, deleted, or updated in the MODEL database and then sending the information to a log file. Use this part of the example script as a guide to setting up appropriate loops to capture record activity and then send this activity to some log file in Listener Perl scripts.

The Listener example script sets up an appropriate loop for capturing record activity in the MODEL database as follows:

```
10 open(LOGFILE, ">>model.log)";
11 while(1){
12  my ($tag, $data) = $snmp->GetResult(-1);
```

```
13  foreach $key (@$data){
14  foreach $rec (keys %$key){
15    {
16   print LOGFILE "$rec : $key->{$rec}","\n";
17    }
18  }
19 }
```

The following list explains the specific numbered items in the previously listed section of the Listener Perl script example:

- **Line 10**

  Calls the `open` operator to open the file handle `LOGFILE` for output (or appending) to the new (or existing) file `model.log`.
- **Lines 11-16**

  Sets up a while loop that executes until all inserted, deleted, and updated database records are processed and written to the `model.log` file.

  Note: In Lines 11-16, the desired information is simply written to a log file (line 16). However, if you want to send the information to a third-party database management application, such as Sybase or Oracle, or a trouble-ticketing system, such as ClearQuest®, use the Perl DBI module. To accomplish this, replace line 16 with the DBI `connect` method and then send the information.

Send database records to different applications

This section of the Listener example script sets up an appropriate loop for "listening" to records inserted, deleted, or updated in the MODEL database and then sending the information to different applications using the Perl DBI module. Use this part of the example script as a guide to setting up appropriate loops to capture record activity and then sending that information to different applications using the Perl DBI module.

The Listener example script sets up an appropriate loop for capturing record activity in the MODEL database and then sending that information to an Oracle database as follows:

```
1 use DBI;
.
.
.
16  $dbname = 'modelEntities'; $user = 'foo';
17  $password = 'foobar'; $dbd = 'Oracle';
18  $dbh = DBI->connect($dbname, $user, $password, $dbd);
19  . . .
20  . . .
21  $dbh->do($statement);
```

The following list explains the specific numbered items in the previously listed section of the Listener Perl script example:

- **Line 1**

  Declares use of the Perl DBI module. See the Perl DBI documentation for more information.
- **Lines 16-21**

  Sets up the appropriate code to send database information to the Oracle database.

Appendix A. RIV Modules Reference

Each RIV module provides constructors and methods used in the Perl scripts that you implement to create custom discovery agent and other client/server applications.

To implement Perl scripts using the RIV modules, you must be familiar with the constructors and methods that each module provides. These constructors and methods are described in manual (reference) page format.

RIV module reference

The RIV module is a container for a set of modules that support implementation of Perl applications on IBM Tivoli Network Manager IP Edition.

The RIV module provides variables, functions, and virtual methods that the Perl API application modules — `RIV::Agent` and `RIV::App` — use.

RIV module synopsis

The `RIV` module synopsis provides summary calls to the variable, functions, and virtual methods that the `RIV::Agent` module and `RIV::App` module can use.

Synopsis

```
# Load the RIV module.
use RIV;

# Call the RIV::RivDebug function.
$RIV::DebugLevel;
RIV::RivDebug($lvl, $debugString);

# Call the RIV::RivMessage function.
$RIV::MessageLevel;
RIV::RivMessage($msglvl, $messageString);

# Call the RIV::RivError function.
RIV::RivError($class, @errorMessageStrings);

# Call the RIV::InputQueueLength function.
$qlen = RIV::InputQueueLength();

# Call the RIV::GetResult function. Note the optional $waitTime
# parameter. Note also that $rivSession stores the application object
# returned in a previous call to the RIV::Agent or RIV::App constructor.
($tag, $value, $domain) = RIV::GetResult($waitTime);
($tag, $value, $domain) = $rivSession->RIV::GetResult([ $waitTime ]);

# Call the RIV::GetResultSet function. Note the optional $waitTime parameter.
$rsKey = $rivSession->RIV::GetResultSet([ $waitTime ]);

# Call the RIV::InputFilter function.
$ok = RIV::InputFilter($pattern, $function);

# Call the PostInput virtual method. Note that $rivSession stores
# the application object returned in a previous call to the RIV::Agent
# or RIV::App constructor.
$ok = $rivSession->PostInput($tag, $data);
# Call the DecryptPassword and EncryptPassword virtual methods.
$txtPwd = RIV::DecryptPassword($encPwd);
```

```
$encPwd = RIV::EncryptPassword($txtPwd);

# Call the RIV::ReadDir function.
$fileListArrayRef = RIV::ReadDir($dirName);

# Call the Latency and Retry Limit virtual methods. Note the optional
# parameters. Note also that $rivSession stores the application object
# returned in a previous call to the RIV::Agent or RIV::App constructor.
$latency = $rivSession->Latency( [$timeoutMilliSeconds] );
$retryLimit = $rivSession->RetryLimit( [$retryLimit] );

# Call the two versions of the PublishMessage virtual methods. Note that
# $rivSession stores the application object returned in a previous call
# to the RIV::Agent or RIV::App constructor.
$ok = $rivSession->PublishMessage($subject, $message);
$ok = $rivSession->PublishMessage($subject, $refHash);

# Call the AddSubject and AddTimer virtual methods. Note that
# $rivsession stores the application object returned in a previous
# call to the RIV::Agent or RIV::App constructor.
$ok = $rivSession->AddSubject($subject, $tag);
$ok = $rivSession->AddTimer($timerval, $tag, $isRepeat);
```

AddSubject

The `AddSubject` virtual method binds the application to the specified message broker subject.

Virtual Method Synopsis

`AddSubject(`*`$subject`*`, `*`$tag`*`)`

Parameters

$subject
Specifies the message broker subject to which `AddSubject` binds the application.

$tag Specifies the tag to be appended to "USER_".

Description

The `AddSubject` virtual method:

- Binds the application to the message broker subject specified in the *$subject* parameter.
- Appends the tag specified in the *$tag* parameter to "USER_". The "USER_*$tag*" messages are returned in a call to the `RIV::GetResult()` function.

The subject is automatically appended with the domain in order to limit messages to purely those for the current domain.

Example Usage

The following example assumes a previous call to the `RIV::App` constructor, which returns a client/server application object to `$app`. A call could also be made to the `RIV::Agent` constructor, which returns a discovery agent application object (typically, to `$agent`).

```
$ok = $app->AddSubject('ITNM/MODEL/NOTIFY', 'model');
```

Returns

Upon completion, the `AddSubject` virtual method returns:

- 0 (zero) — The attempt to bind the application to the message broker subject was unsuccessful.
- 1 — The attempt to bind the application to the message broker subject was successful.

See Also

- "RIV::Agent Constructor" on page 73
- "RIV::App Constructor" on page 95
- "RIV::RivDebug" on page 70
- "RIV::GetResult" on page 63

AddTimer

The `AddTimer` virtual method creates a single-shot or repeating timer.

Virtual Method Synopsis

`AddTimer(`*`$timerVal`*`,` *`$tag`*`,` *`$isRepeat`*`)`

Parameters

$timerVal
: Specifies the time interval, in milliseconds, between timer events.

$tag
: Specifies the tag to be appended to "USER_".

$isRepeat
: Specifies the type of timer to create. The value 0 (zero) creates a single-shot timer and the value 1 creates a repeating timer.

Description

The `AddTimer` virtual method:

- Creates a single-shot or repeating timer, depending on the value passed to the *$isRepeat* parameter. The value of the *$timerVal* parameter specifies the interval, in milliseconds, between the timer events.
- Appends the tag specified in the *$tag* parameter to "USER_". The timer events are returned to the Perl application as "USER_$tag" messages through a call to the `RIV::GetResult` function.

Example Usage

The following example assumes a previous call to the `RIV::App` constructor, which returns a client/server application object to `$app`. A call could also be made to the `RIV::Agent` constructor, which returns a discovery agent application object (typically, to `$agent`).

```
$ok = $app->AddTimer(100, "TIMER", 1);
```

Returns

Upon completion, the `AddTimer` virtual method returns:

- 0 (zero) — The attempt to create a single-shot or repeating timer was unsuccessful.

- 1 — The attempt to create a single-shot or repeating timer was successful.

See Also

- "RIV::Agent Constructor" on page 73
- "RIV::App Constructor" on page 95
- "RIV::RivDebug" on page 70
- "RIV::GetResult" on page 63

DebugLevel

The `RIV` module provides access to the global Network Manager IP Edition debugging level setting through the `$RIV::DebugLevel` variable.

Variable Synopsis

```
$RIV::DebugLevel
```

Description

The `RIV` module provides access to the global Network Manager IP Edition debugging level setting through the `$RIV::DebugLevel` variable. Changing the value of this variable will affect all debugging output. The default value is 0 (zero).

Typically, you use this variable with the following `RIV` module function:

- `RIV::RivDebug`

See Also

- "RIV::RivDebug" on page 70

DecryptPassword

The `DecryptPassword` virtual method decrypts the specified encrypted password.

Synopsis

`DecryptPassword(`*`$encPwd`*`)`

Parameters

$encPwd
Specifies the encrypted password to be decrypted.

Description

The `DecryptPassword` virtual method decrypts the encrypted password specified in the *$encPwd* parameter. You previously encrypted this password in a call to the `EncryptPassword` virtual method. Note that the encryption key must be the same as that used to encrypt the original password.

Example Usage

```
$txtPwd = RIV::DecryptPassword($encPwd);
```

Returns

Upon completion, the `DecryptPassword` virtual method returns the plain text password or `undef` if an error occurred.

See Also

- "EncryptPassword"
- "RIV::Agent Constructor" on page 73
- "RIV::App Constructor" on page 95

EncryptPassword

The `EncryptPassword` virtual method returns an encrypted representation of the specified password.

Synopsis

```
EncryptPassword($txtPwd)
```

Parameters

$txtPwd
Specifies the plain text password to be encrypted.

Description

The `EncryptPassword` virtual method encrypts the plain text password specified in the *$txtPwd* parameter.

Example Usage

```
$encPwd = RIV::EncryptPassword($txtPwd);
```

Returns

Upon completion, the `EncryptPassword` virtual method returns an encrypted and ASCII encoded representation of the specified plain text password or `undef` if an error occurred.

See Also

- "DecryptPassword" on page 56
- "RIV::Agent Constructor" on page 73
- "RIV::App Constructor" on page 95

Latency

The `Latency` virtual method retrieves the timeout for queries.

Virtual Method Synopsis

```
Latency([$timeoutMilliseconds])
```

Parameters

$timeoutMilliseconds
Specifies the timeout, in milliseconds, for the queries. This is an optional parameter and if omitted (that is, no timeout is specified) it returns the value of the timeout.

Description

The `Latency` virtual method:

- Sets a timeout, in milliseconds, for queries if a timeout value is passed to the *$timeoutMilliseconds* parameter. The value passed to *$timeoutMilliseconds* cannot be the `undef` value.
- Returns the timeout, in milliseconds, for queries if the *$timeoutMilliseconds* parameter is omitted (that is, no timeout is specified). If an acknowledgement is not received within this time, further requests are sent up to the retry limit. (The retry limit is specified in a call to the `RetryLimit` virtual method). If no acknowledgement is received after (retry*timeout), an error is returned to the caller.

Example Usage

The following examples assume a previous call to the `RIV::App` constructor, which returns a client/server application object to `$app`. A call could also be made to the `RIV::Agent` constructor, which returns a discovery agent application object (typically, to `$agent`).

The following example shows a call with no *$timeoutMilliseconds* parameter specified:

```
$latency = $app->Latency();
```

The following example shows a call with the *$timeoutMilliseconds* parameter specified:

```
$app->Latency(1000);
```

Returns

Upon completion, the `Latency` virtual method returns the timeout, in milliseconds, if the *$timeoutMilliseconds* parameter is omitted.

See Also

- "RIV::Agent Constructor" on page 73
- "RIV::App Constructor" on page 95
- "RetryLimit" on page 61
- "RIV::RivDebug" on page 70

MessageLevel

The `RIV` module provides access to the global Network Manager IP Edition logging level setting through the `$RIV::MessageLevel` variable.

Variable Synopsis

```
$RIV::MessageLevel
```

Description

The `RIV` module provides access to the global Network Manager IP Edition logging setting through the `$RIV::MessageLevel` variable. Changing the value of this variable will affect all logging output.

Typically, you use this variable with the following `RIV` module function:

- `RIV::RivMessage`

See Also

- "RIV::RivMessage" on page 71

PostInput

The `PostInput` virtual method adds a message to the queue.

Virtual Method Synopsis

```
PostInput($tag, $data)
```

Parameters

$tag Specifies the tag to be associated with the input message.

$data Specifies the data in the message.

Description

The `PostInput` virtual method adds the input message specified in the *$data* parameter to the queue along with the associated tag specified in the *$tag* parameter.

Example Usage

The following example assumes a previous call to the `RIV::App` constructor, which returns a client/server application object to `$app`. A call could also be made to the `RIV::Agent` constructor, which returns a discovery agent application object (typically, to `$agent`).

```
$app = RIV::App::new(.......);
$app->PostInput("myTag", "test data");
```

Returns

Upon completion, the `PostInput` virtual method returns:

- 0 (zero) — The attempt to add the input message to the queue was unsuccessful.
- 1 — The attempt to add the input message to the queue was successful.

See Also

- "RIV::Agent Constructor" on page 73
- "RIV::App Constructor" on page 95
- "RIV::RivDebug" on page 70

PublishMessage

The `PublishMessage` virtual method publishes the specified message string.

Virtual Method Synopsis

```
PublishMessage($subject, $message)
```

Parameters

$subject
Specifies the unqualified Network Manager IP Edition subject used for message broker messaging.

$message
Specifies a valid ASCII message string.

Description

The `PublishMessage` virtual method publishes the message string specified in the *$message* parameter on the subject specified in the *$subject* parameter. The value specified in *$subject* must be unqualified, that is, it must be without the **.DOMAIN** suffix. The message in *$message* must be a valid ASCII string.

Example Usage

The following example assumes a previous call to the `RIV::App` constructor, which returns a client/server application object to `$app`. A call could also be made to the `RIV::Agent` constructor, which returns a discovery agent application object (typically, to `$agent`).

```
$ok = $app->PublishMessage('ITNM/MODEL/QUERY', "hello");
```

Returns

Upon completion, the `PublishMessage` virtual method returns:
- 0 (zero) — The attempt to publish the specified message was unsuccessful.
- 1 — The attempt to publish the specified message was successful.

See Also
- "RIV::Agent Constructor" on page 73
- "RIV::RivDebug" on page 70
- "PublishMessage"

PublishMessage

The `PublishMessage` virtual method encodes the hash reference into a message broker message.

Virtual Method Synopsis

```
PublishMessage($subject, $refHash)
```

Parameters

$subject
Specifies the unqualified Network Manager IP Edition subject used for message broker messaging.

$refHash
Specifies a reference to a hash that contains the message to be sent.

Description

The `PublishMessage` virtual method encodes the hash specified in the `$refHash` parameter into a message broker message and publishes it on the subject specified in the `$subject` parameter. The hash passed to `$refHash` may be nested. The value passed to the `$subject` parameter must be unqualified, that is, it must be without the **.DOMAIN** suffix.

Note: The message type and contents must be what the consumer is expecting. There is a chance that the core processes will SIGSEGV if they receive unexpected data (for example, publishing a string message to a `NOTIFY` subject).

Example Usage

The following example assumes a previous call to the `RIV::App` constructor, which returns a client/server application object to `$app`. A call could also be made to the `RIV::Agent` constructor, which returns a discovery agent application object (typically, to `$agent`).

```
my %gg; $gg{'foo'} = 'bar'; $gg{'color'} = 'red';

$ok = $app->PublishMessage('ITNM/MODEL/QUERY', \%gg);
```

Returns

Upon completion, the `PublishMessage` virtual method returns:

- 0 (zero) — The attempt to encode the hash and publish the specified message was unsuccessful.
- 1 — The attempt to encode the hash and publish the specified message was successful.

See Also

- "RIV::Agent Constructor" on page 73
- "RIV::RivDebug" on page 70
- "PublishMessage" on page 59

RetryLimit

The `RetryLimit` virtual method sets the retry limit for queries or returns the maximum number of retries for queries.

Virtual Method Synopsis

```
RetryLimit([$retryLimit])
```

Parameters

$retryLimit
Specifies the retry limit for a specified query. This is an optional parameter and if omitted (that is, no retry limit is specified) it returns the maximum number of retries.

Description

The `RetryLimit` virtual method:

- Sets a retry limit for queries if a value is passed to the *$retryLimit* parameter.
- Returns the maximum number of retries for a specified query if the *$retryLimit* parameter is omitted (that is, no retry limit is specified).

Example Usage

The following examples assume a previous call to the `RIV::App` constructor, which returns a client/server application object to `$app`. A call could also be made to the `RIV::Agent` constructor, which returns a discovery agent application object (typically, to `$agent`).

The following usage example shows a call with no *$retryLimit* parameter specified:

```
$retry = $app->RetryLimit();
```

The following usage example shows a call with the *$retryLimit* parameter specified:

```
$app->RetryLimit(5);
```

Returns

Upon completion, the `RetryLimit` virtual method returns the maximum number of retries for a specified query if the *$retryLimit* parameter is omitted.

See Also

- "RIV::Agent Constructor" on page 73
- "RIV::RivDebug" on page 70

RIV::GetInput

The `RIV::GetInput` function has been deprecated by `RIV::GetResult`. The documentation exists for historical purposes only.

Synopsis

```
RIV::GetInput($waitTime [,$pattern])
```

Parameters

$waitTime
: Specifies the time, in seconds, to wait before returning. If *$waitTime* is negative, `RIV::GetInput` waits forever for the response.

$pattern
: Specifies the pattern of tags that the user is interested in. Only messages with a matching tag will be returned. All other messages will be left in the queue for retrieval at a later time. This parameter is optional.

Description

The `RIV::GetInput` function provides a mechanism for synchronizing a single-threaded Perl application with the multithreaded Network Manager IP Edition platform. There is currently no support for direct interface with Perl threads.

The normal usage of `RIV::GetInput` takes a single parameter to specify the number of seconds to wait for input before returning. A value of 0 (zero) means "do not wait", and a negative value means "wait forever".

Because Network Manager IP Edition platform receives input either directly or indirectly through message broker, the input data is placed on a FIFO together with its identifying tag. One input item is returned for each call to `RIV::GetInput`. Items are returned as an array of size 3 containing the item tag, item tag value, and the application domain (that is, the domain string specified in the call to `RIV::App::new`). For example:

```
($tag, $data, $domain) = RIV::GetInput(-1);
```

If there is only one active `RIV::App`, the domain value may be ignored. However, if multiple `RIV::App` objects have been created, the value of *$domain* must be used to determine the source of the input.

Value types depend on the item returned and must be interpreted in the context of the value of *$tag*. Tag values are either specified in a call to create the input stream or are from a set of standard tags. User specified tags are returned from `RIV::GetInput` with the prefix `USER_`. Standard tags include:

- QUERY — (`RIV::OQL` query results)
- UPDATE — (`RIV::OQL` updates)
- NE — (`RIV::Agent`)
- TIMEOUT — (all - wait time exceeded and no data)

The extended form of `RIV::GetInput` uses a second parameter (*$pattern*) to specify a regular expression pattern for matching against input tag strings. Only data items with matching tags will be returned. This form is useful for temporarily suspending delivery of input to all but the wanted channel and has the effect of taking input data items out of turn. Non-matching input tags are kept in the queue and will be delivered in sequence when the standard form of `RIV::GetInput` is used, or a matching pattern is specified to a subsequent call of the extended version.

Example Usage

```
($tag, $data, $domain) = RIV::GetInput(-1);
```

Returns

Upon successful completion, the `RIV::GetInput` function returns:

- *$tag* — The tag associated with the message.
- *$data* — The data associated with the tag. The *$data* value could be a string or a reference to any data structure and will be interpreted based on the *$tag* value.
- *$domain* — The domain name. This return will only be of interest if multiple domains are running.

See Also

- "RIV::GetResult"

RIV::GetResult

The `RIV::GetResult` function provides the "standard" mechanism for synchronizing a single-threaded Perl application with the multi-threaded Network Manager IP Edition platform. There is currently no support for a direct interface with Perl threads.

Synopsis

```
RIV::GetResult([$waitTime])
```

Parameters

$waitTime
Specifies the time, in seconds, to wait for input before returning. If *$waitTime* is negative, `RIV::GetResult` waits forever for the response. This is an optional parameter that defaults to the latency of the application.

Description

The typical call to the `RIV::GetResult` function takes a single parameter to specify the number of seconds to wait for input before returning. A value of 0 (zero) means "do not wait" and a value of minus one (-1) means "wait forever". The *$waitTime* parameter is optional and, if it is not specified, it defaults to the latency associated with the application.

As the Network Manager IP Edition platform receives input (either directly or indirectly through message broker), the input data is placed on a FIFO basis, together with its identifying tag. One input item is returned for each call to `RIV::GetResult`. Items are returned as an array of size 3, containing the item tag, its value and the application domain (that is, the domain string specified in a call to the `RIV::App::new()` constructor). For example:

```
my ($tag, $data, $domain) = $app->RIV::GetResult(-1);
```

If there is only one active `RIV::App`, the domain value may be ignored. However, if multiple `RIV::App` objects have been created, the value of *$domain* must be used to determine the source of the input.

Value types depend on the item returned and must be interpreted in the context of the value of *$tag*. Tag values are either specified in a call to create the input stream or are from a set of standard tags. User specified tags are returned from `RIV::GetResult()` with the prefix "USER_". The following example identifies the standard tags:

```
OQLQuery  (RIV::OQL query results)
OQLUpdate (RIV::OQL updates)
NE        (RIV::Agent)
TIMEOUT   (all - wait time exceeded and no data)
```

Example Usage

The following example assumes a previous call to the `RIV::App` constructor, which returns a client/server application object to `$app`. A call could also be made to the `RIV::Agent` constructor, which returns a discovery agent application object (typically, to `$agent`).

```
($tag, $data, $domain) = $app->RIV::GetResult();
($tag, $data, $domain) = $app->RIV::GetResult(10);
($tag, $data, $domain) = $app->RIV::GetResult(-1);
```

Returns

Upon successful completion, the `RIV::GetResult` function returns:

- *$tag* — The tag associated with the message.
- *$data* — The data associated with the tag. The *$data* value could be a string or a reference to any data structure and will be interpreted based on the *$tag* value.
- *$domain* — The domain name. This return will only be of interest if multiple domains are running.

See Also

- “RIV::Agent Constructor” on page 73
- “RIV::App Constructor” on page 95
- “RIV::GetInput” on page 62
- “RIV::RivDebug” on page 70

RIV::InputFilter

The `RIV::InputFilter` function binds the function referenced by *$function* to input tags matching the regular expression *$pattern*.

Synopsis

```
RIV::InputFilter($pattern [,$function]
[,$arg])
```

Parameters

$pattern
: Specifies the tag corresponding to the regular expression to which the filter must be run.

$function
: Specifies the function that will be executed if the input tag matches the regular expression passed to the *$pattern* parameter. This parameter is optional. If no function is specified, the existing callback for the pattern is deleted.

$arg
: Specifies an argument to be passed to the function specified in the *$function* parameter. This parameter is optional.

Description

The `RIV::InputFilter` function binds the function referenced by *$function* to input tags matching the regular expression *$pattern*. Whenever the application program calls the `RIV::GetResult` function and data with a matching tag is returned, the corresponding function is called instead of a return from `RIV::GetResult`. If all input tags match one of the patterns passed to `RIV::InputFilter`, the effect is as if the original call to `RIV::GetResult` never returned. The called function must not call `RIV::GetResult`. Calling the `RIV::InputFilter` function with a value of `undef` for *$function* removes the filter.

Example Usage

```
$ok = RIV::InputFilter("NE", $method1);
```

Returns

Upon completion, the `RIV::InputFilter` function returns:

- 0 (zero) — The attempt to bind the function referenced by *$function* was unsuccessful.
- 1 — The attempt to bind the function referenced by *$function* was successful.

See Also

- "RIV::Agent Constructor" on page 73
- "RIV::App Constructor" on page 95
- "RIV::GetResult" on page 63
- "RIV::RivDebug" on page 70

RIV::InputQueueLength

The `RIV::InputQueueLength` function returns the number of items waiting in the application's input queue.

Synopsis

```
RIV::InputQueueLength()
```

Parameters

None

Description

The `RIV::InputQueueLength` function returns the number of items waiting in the application's input queue, that is, the number of times `RIV::GetResult` would need to be called in order to drain the queue.

Example Usage

```
$queue_length = RIV::InputQueueLength();
```

Returns

Upon successful completion, the `RIV::InputQueueLength` function returns the number of items waiting in the application's input queue.

See Also

- "RIV::Agent Constructor" on page 73
- "RIV::App Constructor" on page 95
- "RIV::GetResult" on page 63
- "RIV::RivDebug" on page 70

RIV::IsIpNotLoopBackOrMulticast

The `RIV::IsIpNotLoopBackOrMulticast` function returns true if the address parameter is a valid IP address, and not a loop back or multicast address.

Synopsis

```
RIV::IsIpNotLoopBackOrMulticast($ipAddress)
```

Parameters

$ipAddress
Specifies the IP address that must be checked for validity.

Description

The `RIV::IsIpNotLoopBackOrMulticast` function returns true if the address passed to the *$ipAddress* parameter is a valid IP address, and not a loop back or multicast address.

Example Usage

```
$result = RIV::IsIpNotLoopBackOrMulticast($ipAddress);
```

Returns

Upon completion, the `RIV::IsIpNotLoopBackOrMulticast` function returns:

- 0 (zero) — The IP address is not valid or the IP address is a loop back or multicast address.
- 1 — The IP address is valid.

See Also

- "RIV::Agent Constructor" on page 73
- "RIV::App Constructor" on page 95
- "RIV::RivDebug" on page 70

RIV::IsIpValid

The `RIV::IsIpValid` function returns true if the address parameter is a valid IP address.

Synopsis

```
RIV::IsIpValid($ipAddress)
```

Parameters

$ipAddress
Specifies the IP address that must be checked for validity.

Description

The `RIV::IsIpValid` function returns true if the address passed to the *$ipAddress* parameter is a valid IP address. More specifically, the function checks if *$ipAddress* is a valid IPv4 or IPv6 address using the functions specific to those address families.

Example Usage

```
$result = RIV::IsIpValid($ipAddress);
```

Returns

Upon completion, the `RIV::IsIpValid` function returns:

- 0 (zero) — The IP address is not valid.
- 1 — The IP address is valid.

See Also

- "RIV::Agent Constructor" on page 73
- "RIV::App Constructor" on page 95
- "RIV::IsIpv4Valid" on page 68
- "RIV::IsIpv6Valid" on page 68
- "RIV::RivDebug" on page 70

RIV::IsIpv4Valid

The `RIV::IsIpv4Valid` function returns true if the address parameter is a valid IPv4 address.

Synopsis

`RIV::IsIpv4Valid(`*`$ipAddress`*`)`

Parameters

$ipAddress
Specifies the IP address that must be checked for validity.

Description

The `RIV::IsIpv4Valid` function returns true if the address passed to the *$ipAddress* parameter is a valid IPv4 address. More specifically, the function checks that the IP address is of the form `a.b.c.d` and that each number in the IP address (`a`, `b`, `c`, `d`) is less than 255.

Example Usage

```
$result = RIV::IsIpv4Valid($ipAddress);
```

Returns

Upon completion, the `RIV::IsIpv4Valid` function returns:

- 0 (zero) — The IP address is not valid.
- 1 — The IP address is valid.

See Also

- "RIV::Agent Constructor" on page 73
- "RIV::App Constructor" on page 95
- "RIV::IsIpValid" on page 67
- "RIV::Ipv6Valid"
- "RIV::RivDebug" on page 70

RIV::IsIpv6Valid

The `RIV::IsIpv6Valid` function returns true if the address parameter is a valid IPv6 address.

Synopsis

`RIV::IsIpv6Valid(`*`$ipAddress`*`)`

Parameters

$ipAddress
Specifies the IP address that must be checked for validity.

Description

The `RIV::IsIpv6Valid` function returns true if the address passed to the *$ipAddress* parameter is a valid IPv6 address. More specifically, the function checks that the IP address is of the standard forms as defined in RFC429.

Example Usage

```
$result = RIV::IsIpv6Valid($ipAddress);
```

Returns

Upon completion, the `RIV::IsIpv6Valid` function returns:
- 0 (zero) — The IP address is not valid.
- 1 — The IP address is valid.

See Also
- "RIV::Agent Constructor" on page 73
- "RIV::App Constructor" on page 95
- "RIV::IsIpValid" on page 67
- "RIV::Ipv4Valid" on page 68
- "RIV::RivDebug" on page 70

RIV::ReadDir

The `RIV::ReadDir` function returns a reference to an array of filenames that reside in the specified directory.

Synopsis

```
RIV::ReadDir($dirName)
```

Parameters

$dirName
Specifies the name of the directory to read.

Description

The `RIV::ReadDir` function returns a reference to an array of filenames that reside in the directory specified in the *$dirName* parameter. The `RIV::ReadDir` function provides the same functionality as the standard Perl `readdir` function. The `RIV::ReadDir` function is supplied to accommodate known issues when trying to use `readdir` with **ncp_perl** on some Linux platforms.

Example Usage

```
$fileListArrayRef = RIV::ReadDir($dirName);
```

Returns

Upon completion, the `RIV::ReadDir` function returns a reference to an array of filenames that reside in the directory specified in the *$dirName* parameter.

See Also
- "RIV::Agent Constructor" on page 73
- "RIV::App Constructor" on page 95

RIV::RivDebug

The `RIV::RivDebug` function prints a list of debug message strings to the standard output.

Synopsis

`RIV::RivDebug(`*`$lvl`*`,`*`@debugMessageStrings`*`)`

Parameters

$lvl Specifies the debug level. Specify a value of 1-4, where 4 represents the most detailed output.

@debugMessageStrings
Specifies the strings to be printed when the debug level is set.

Description

The `RIV::RivDebug` function prints the space-concatenated list of strings from the *@debugMessageStrings* parameter to the standard output if the value of the `$RIV::DebugLevel` global variable is equal to, or greater than, the value specified in the *$lvl* parameter.

Example Usage

```
RIV::RivDebug(4, "my debug message here");
```

Returns

Upon completion, the `RIV::RivDebug` function returns no records or values.

See Also

- "DebugLevel" on page 56

RIV::RivError

The `RIV::RivError` function provides a convenient way to display error messages.

Synopsis

`RIV::RivError(`*`$class`*`, `*`@errorMessageStrings`*`)`

Parameters

$class Specifies the name of the calling Perl API module (for example, `RIV::App`).

@errorMessageStrings
Specifies the error message string to be printed when an error occurs.

Description

The `RIV::RivError` function prints the error messages tagged with the name of the calling class (`RIV::*`) and will integrate with the forthcoming trace package.

Example Usage

```
RIV::RivError("RIV::App", "An error has occurred");
```

Returns

Upon completion, the `RIV::RivError` function returns no records or values.

See Also

- "RIV::Agent Constructor" on page 73
- "RIV::App Constructor" on page 95
- "RIV::RivDebug" on page 70

RIV::RivMessage

The `RIV::RivMessage` function prints a list of log message strings to the standard output.

Synopsis

`RIV::RivMessage(`*`$msglvl`*`,`*`@messageLevelStrings`*`)`

Parameters

$msglvl
Specifies the level of messages to be logged (the default is warn):
- debug
- info
- warn
- error
- fatal

@messageLevelStrings
Specifies the strings to be printed when the logging level is set.

Description

The `RIV::RivMessage` function prints the space-concatenated list of strings from the *@messageLevelStrings* parameter to the standard output if the value of the `$RIV::MessageLevel` global variable is equal to, or greater than, the value specified in the *$msglvl* parameter.

Example Usage

```
RIV::RivMessage("warn", "my messagelevel message here");
```

Returns

Upon completion, the `RIV::RivMessage` function returns no records or values.

See Also

- "MessageLevel" on page 58

RIV::Agent module reference

The `RIV::Agent` module enables developers to implement Network Manager discovery agents.

The `RIV::Agent` module provides a constructor and the following categories of methods:
- SNMP operation methods
- DNS operation methods
- Ping operation methods
- IP and MAC address operation methods

- Telnet operation methods
- Network entity operation methods
- Threads methods

RIV::Agent module synopsis

The `RIV::Agent` module synopsis provides summary synopses of the constructor and methods that discovery agents use.

```
# Load the RIV::Agent module
use RIV::Agent;

# Call the RIV::Agent constructor and return a RIV::Agent object
$agent = RIV::Agent::new($param, $agentName);

# Call the SNMP operation methods
$varOp = $agent->SnmpGet($ne, $oid, $instance, $communitySuffix);
$varOpArray = $agent->SnmpGetNext($ne, $oid, $instance, $communitySuffix);
$varOpArray = $agent->SnmpGetBulk($ne, $oidList, $nonRepeaters,
                                  $maxRepeaters, $communitySuffix);

# Call the DNS operation methods
$refAllIpAddrs = $agent->GetDNSAllIpAddrs($name);
$refAllNames = $agent->GetDNSAllNames($ipAddress);
$ip = $agent->GetDNSFirstIpAddr($name);
$name = $agent->GetDNSFirstName($ipAddress);

# Call the IP and MAC address operation methods
$ip = $agent->GetIpArp($macAddress);
$mac = $agent->GetMacArp($ipAddress);
$routelist = $agent->GetTraceRoute($ipAddress, $protocol);

# Call the ping operation methods
$reply = $agent->GetPingIP($ipAddress, $protocol);
$replylist = $agent->GetPingList($ipAddressList, $protocol);
$reply = $agent->GetPingSubnet($subnet, $netMask, $protocol);
$agent->PingIP($ipAddress, $protocol);
$agent->PingList($ipAddressList, $protocol);
$agent->PingSubnet($subnet, $netMask, $protocol);

# Call the Telnet operation methods
$telarray = $agent->GetMultTelnet($ne, $commandList);
$teldata = $agent->GetTelnet($ne, $command, $regExp);
$teldata = $agent->GetTelnetCols($ne, $command, $regExpList, $colNameList);

# Call the Network Entity operation methods
$agent->SendNEToDisco($NE);
$agent->SendNEToNextPhase($NE);

# Call the threads operation methods
$agent->LockThreads();
$agent->UnLockThreads();
```

RIV::Agent Constructor

The `RIV::Agent` constructor creates a Network Manager discovery agent with the specified name.

Constructor

`new(`*`$param`*`, `*`$agentName`*`)`

Parameters

$param
: Specifies a `RIV::Param` object that was returned in a previous call to the `RIV::Param` constructor.

$agentName
: Specifies a string that identifies the name of the discovery agent to be created in the domain specified by the `RIV::Param` object passed to the *$param* parameter.

Description

The `RIV::Agent` constructor creates a Network Manager discovery agent with an agent name as specified in the *$agentName* parameter. This agent name resides in the domain as specified by the *$param* parameter (that is, a `RIV::Param` object).

The `RIV::Agent` constructor uses the Transmission Control Protocol (TCP) to establish the necessary connections to the Discovery Server and Helper Server. To ensure that the databases for the discovery agent are created inside the Discovery Server, the `$agentName.agnt` file must be defined in the `$NCHOME/disco/agents` directory before the Discovery Engine executable, `ncp_disco`, is started.

Example Usage

The following example:
- Calls the `RIV::Param` constructor and stores the return value (a `RIV::Param` object) in the *$param* variable.
- Calls the `RIV::Agent` constructor and specifies a discovery agent name of `foo_perl_disco_agent`.
- Stores the return value (a `RIV::Agent` object) in the *$agent* variable.

```
$param = new RIV::Param();
$agent = new RIV::Agent($param, "foo_perl_disco_agent");
```

Returns

Upon completion, the `RIV::Agent` constructor returns a `RIV::Agent` object. This object is associated with the discovery agent specified in the *$agentName* parameter.

GetDNSAllIpAddrs

The `GetDNSAllIpAddrs` method returns all IP addresses corresponding to a particular node name.

Method Synopsis

`GetDNSAllIpAddrs(`*`$name`*`)`

Parameters

$name Specifies the name of the node whose corresponding IP addresses are of interest.

Description

The `GetDNSAllIpAddrs` method returns all IP addresses corresponding to the node name specified in the *$name* parameter.

Notes

The `GetDNSAllIpAddrs` method issues the appropriate DNS request to the Helper Server, which performs the actual work. Thus, the Helper Server (and ncp_ctrl) must be running so that this method can make the appropriate DNS request.

Example Usage

The following example:

- Assumes a previous call to the `RIV::Agent` constructor, which returns a `RIV::Agent` object (represented by `$agent->`).
- Returns to the *$refAllIpAddrs* variable a reference to an array that contains all IP addresses corresponding to the node called `foo`.
- Calls the `print` operator to send each IP address in the list to standard output.

```
$refAllIpAddrs = $agent->GetDNSAllIpAddrs("foo");
print @$refAllIpAddrs;
```

Returns

Upon completion, the `GetDNSAllIpAddrs` method returns a reference to an array of IP addresses corresponding to the specified node name.

GetDNSAllNames

The `GetDNSAllNames` method returns all node names corresponding to a specific IP address.

Method Synopsis

`GetDNSAllNames(`*`$ipAddress`*`)`

Parameters

$ipAddress
Specifies the IP address whose corresponding node names are of interest.

Description

The `GetDNSAllNames` method returns all node names corresponding to the IP address specified in the *$ipAddress* parameter. by issuing a DNS request to the Helper Server.

Notes

The `GetDNSAllNames` method issues the appropriate DNS request to the Helper Server, which performs the actual work. Thus, the Helper Server (and ncp_ctrl) must be running so that this method can make the appropriate DNS request.

Example Usage

The following example:
- Assumes a previous call to the `RIV::Agent` constructor, which returns a `RIV::Agent` object (represented by `$agent->`).
- Returns to the *$refAllNames* variable a reference to an array that contains all node names corresponding to the IP address `1.2.3.4`.
- Calls the `print` operator to send each node name in the list to standard output.

```
$refAllNames = $agent->GetDNSAllNames("1.2.3.4");
print @$refAllNames;
```

Returns

Upon completion, the `GetDNSAllNames` method returns a reference to an array of names corresponding to a specific IP address.

GetDNSFirstIpAddr

The `GetDNSFirstIpAddr` method returns the first IP address in the list of IP addresses for the specified node.

Method Synopsis

`GetDNSFirstIpAddr(`*`$name`*`)`

Parameters

$name Specifies the name of the node whose first IP address in the corresponding list of IP addresses is of interest.

Description

The `GetDNSFirstIpAddr` method returns the first IP address in the list of IP addresses corresponding to the node specified in the *$name* parameter.

Notes

The `GetDNSFirstIpAddr` method issues the appropriate DNS request to the Helper Server, which performs the actual work. Thus, the Helper Server (and ncp_ctrl) must be running so that this method can make the appropriate DNS request.

Example Usage

The following example:

- Assumes a previous call to the `RIV::Agent` constructor, which returns a `RIV::Agent` object (represented by `$agent->`).
- Returns to the *$ip* variable the first IP address in the list of IP addresses corresponding to the node called `foo`.

```
$ip = $agent->GetDNSFirstIpAddr("foo");
```

Returns

Upon completion, the `GetDNSFirstIpAddr` method returns the first IP address in the list of IP addresses for the specified node. This IP address is a scalar value.

GetDNSFirstName

The `GetDNSFirstName` method returns the first node name in the list of node names for the specified IP address.

Method Synopsis

`GetDNSFirstName(`*`$ipAddress`*`)`

Parameters

$ipAddress
: Specifies the IP address whose first node name in the corresponding list of node names is of interest.

Description

The `GetDNSFirstName` method returns the first node name in the list of node names corresponding to the IP address specified in the *$ipAddress* parameter.

Notes

The `GetDNSFirstName` method issues the appropriate DNS request to the Helper Server, which performs the actual work. Thus, the Helper Server (and ncp_ctrl) must be running so that this method can make the appropriate DNS request.

Example Usage

The following example:

- Assumes a previous call to the `RIV::Agent` constructor, which returns a `RIV::Agent` object (represented by `$agent->`).
- Returns to the *$name* variable the first node name in the list of node names corresponding to the IP address `1.2.3.1`.

```
$name = $agent->GetDNSFirstName("1.2.3.1");
```

Returns

Upon completion, the `GetDNSFirstName` method returns the first node name in the list of node names for the specified IP address. This node name is a scalar value.

GetIpArp

The `GetIpArp` method converts the specified MAC address to its corresponding IP address.

Method Synopsis

```
GetIpArp($macAddress)
```

Parameters

$macAddress
Specifies the MAC address to be converted to its corresponding IP address.

Description

The `GetIpArp` method converts the MAC address specified in the *macAddress* parameter to its corresponding IP address.

Notes

The `GetIpArp` method issues the appropriate ARP request to the Helper Server, which performs the actual work. Thus, the Helper Server (and ncp_ctrl) must be running so that this method can make the appropriate ARP request.

Example Usage

The following example:

- Assumes a previous call to the `RIV::Agent` constructor, which returns a `RIV::Agent` object (represented by `$agent->`).
- Returns to the *$ip* variable the IP address corresponding to the MAC address `00-0C-F1-56-98-AD`.

```
$ip = $agent->GetIpArp("00-0C-F1-56-98-AD");
```

Returns

Upon completion, the `GetIpArp` method returns the IP address corresponding to the specified MAC address.

GetMacArp

The `GetMacArp` method converts the specified IP address to a MAC address.

Method Synopsis

```
GetMacArp($ipAddress)
```

Parameters

$ipAddress
Specifies the IP address to be converted to an associated MAC address.

Description

The `GetMacArp` method converts the IP address specified in the *ipAddress* parameter to an associated MAC address.

Notes

The `GetMacArp` method issues the appropriate ARP request to the Helper Server, which performs the actual work. Thus, the Helper Server (and ncp_ctrl) must be running so that this method can make the appropriate ARP request.

Example Usage

The following example:

- Assumes a previous call to the `RIV::Agent` constructor, which returns a `RIV::Agent` object (represented by `$agent->`).
- Returns to the *$macAddress* variable the MAC address corresponding to the IP address `1.2.3.1`.

```
$macAddress = $agent->GetMacArp("1.2.3.1");
```

Returns

Upon completion, the `GetMacArp` method returns the MAC address corresponding to the specified IP address.

GetMultTelnet

The `GetMultTelnet` method initiates a Telnet session on the specified network device and then executes the specified Telnet commands on that network device.

Method Synopsis

`GetMultTelnet(`*`$ne`*`, `*`$commandList`*`)`

Parameters

$ne
Specifies a reference to the network entity, in this case the network device on which to execute the Telnet commands specified in the `$commandList` parameter.

$commandList
Specifies an array that contains the Telnet commands to execute on the network device specified in the `$ne` parameter.

Description

The `GetMultTelnet` method initiates a Telnet session on the network device specified in the `$ne` parameter. It then executes the Telnet commands specified in the `$commandList` parameter on that network device.

Notes

The `GetMultTelnet` method issues the appropriate Telnet request to the Helper Server, which performs the actual work. Thus, the Helper Server (and ncp_ctrl) must be running so that this method can make the appropriate Telnet request.

Returns

Upon completion, the `GetMultTelnet` method returns an array of data corresponding to each Telnet command executed during the Telnet session.

GetPingIP

The `GetPingIP` method issues a ping at the specified IP address to determine if a network device exists at that address.

Method Synopsis

```
GetPingIP($ipAddress [, $protocol])
```

Parameters

$ipAddress
Specifies the IP address to be pinged.

$protocol
Specifies an optional parameter that identifies the IP protocol. You can specify one of the following values:

- `1` — Specifies Internet Protocol version 4 (IPv4).
- `3` — Specifies Internet Protocol version 6 (IPv6).

Description

The `GetPingIP` method pings the IP address specified in the *ipAddress* parameter. A network device that exists at the specified IP address will respond to this ping request.

Notes

The `GetPingIP` method issues the appropriate ping request to the Helper Server, which performs the actual work. Thus, the Helper Server (and ncp_ctrl) must be running so that this method can make the appropriate ping request.

Example Usage

The following example:

- Assumes a previous call to the `RIV::Agent` constructor, which returns a `RIV::Agent` object (represented by `$agent->`).
- Returns to the *$device_exists* variable a value of 0 (zero) or 1.

```
$device_exists = $agent->GetPingIP("1.2.3.1");
```

Returns

Upon completion, the `GetPingIP` method returns one of the following values:

- 0 (zero) — There is no network device at the specified IP.
- 1 — There is a network device at the specified IP address.

GetPingList

The `GetPingList` method issues a ping at the specified list of IP addresses to determine if network devices exist at those addresses.

Method Synopsis

`GetPingList(`*`$ipAddressList`*` [, $protocol])`

Parameters

$ipAddressList
: Specifies the list of IP addresses to be pinged.

$protocol
: Specifies an optional parameter that identifies the IP protocol. You can specify one of the following values:

- `1` — Specifies Internet Protocol version 4 (IPv4).
- `3` — Specifies Internet Protocol version 6 (IPv6).

Description

The `GetPingList` method pings the list of IP addresses specified in the *ipAddressList* parameter. Network devices that exist at the specified IP addresses will respond to this ping request.

Notes

The `GetPingList` method issues the appropriate ping request to the Helper Server, which performs the actual work. Thus, the Helper Server (and ncp_ctrl) must be running so that this method can make the appropriate ping request.

Returns

Upon completion, the `GetPingList` method returns a list that identifies whether the network devices exist at the specified IP addresses. The following values are specified in the list:

- 0 (zero) — There is no network device at the specified IP.
- 1 — There is a network device at the specified IP address.

GetPingSubnet

The `GetPingSubnet` method pings the specified subnet and returns whether a reply was received. issues a ping at the specified subnet to determine if one or more network devices exist at that subnet.

Method Synopsis

`GetPingSubnet(`*`$subnet`*`, `*`$netMask`*` [, $protocol])`

Parameters

$subnet
: Specifies the IP address of the subnet to be pinged. Typically, subnets are defined as all devices whose IP addresses have the same prefix. Thus, all devices with IP addresses that start with `1.1.1` would be part of the same subnet.

$netmask
Specifies the mask used to determine the subnet to which an IP address belongs.

$protocol
Specifies an optional parameter that identifies the IP protocol. You can specify one of the following values:

- `1` — Specifies Internet Protocol version 4 (IPv4).
- `3` — Specifies Internet Protocol version 6 (IPv6).

Description

The `GetPingSubnet` method pings the subnet specified in the *subnet* parameter Network devices that exist at the specified subnet will respond to this ping request.

Notes

The `GetPingSubnet` method issues the appropriate ping request to the Helper Server, which performs the actual work. Thus, the Helper Server (and ncp_ctrl) must be running so that this method can make the appropriate ping request.

Returns

Upon completion, the `GetPingSubnet` method returns a list that identifies whether the network devices exist at the specified subnet. The following values are specified in the list:

- 0 (zero) — There is no network device at the specified IP.
- 1 — There is a network device at the specified IP address.

GetTelnet

The `GetTelnet` method initiates a Telnet session on the specified network device and executes the specified Telnet command on that network device.

Method Synopsis

`GetTelnet(`*`$ne`*`, `*`$command`*`, `*`$regExp`*`)`

Parameters

$ne Specifies a reference to the network entity, in this case the network device on which to execute the Telnet command specified in the `$command` parameter.

$command
Specifies the Telnet command to execute on the network device specified in the `$ne` parameter.

$regExp
Specifies the regular expression to apply to the response of the Telnet command specified in the `$command` parameter.

Description

The `GetTelnet` method initiates a Telnet session on the network device specified in the `$ne` parameter. The `GetTelnet` method then executes the Telnet command specified in the `$command` parameter on that network device.

Notes

The `GetTelnet` method issues the appropriate Telnet request to the Helper Server, which performs the actual work. Thus, the Helper Server (and ncp_ctrl) must be running so that this method can make the appropriate Telnet request.

Returns

Upon completion, the `GetTelnet` method returns the data corresponding to the Telnet command executed during the Telnet session. This data must meet the regular expression supplied in the *$regExp* parameter.

GetTelnetCols

The `GetTelnetCols` method initiates a Telnet session on the specified network device and executes the specified Telnet command on that network device.

Method Synopsis

`GetTelnetCols(`*`$ne`*`, `*`$command`*`, `*`$regExpList`*`, `*`$colNameList`*`)`

Parameters

$ne Specifies a reference to the network entity, in this case the network device on which to execute the Telnet command specified in the `$command` parameter.

$command
Specifies the Telnet command to execute on the network device specified in the `$ne` parameter.

$regExpList
Specifies an array of regular expressions to apply to the response of the Telnet command specified in the `$command` parameter.

$colNameList
Specifies an array of table columns.

Description

The `GetTelnetCols` method:

- Initiates a Telnet session on the network device specified in the `$ne` parameter by issuing a Telnet request through the Helper Server.
- Executes the Telnet command specified in the `$command` parameter on that network device.
- Splits data into columns based on the regular expression specified in the `$regExpList` parameter. This method is particularly suited to responses to Telnet commands that consist of tables.

Notes

The `GetTelnetCols` method issues the appropriate Telnet request to the Helper Server, which performs the actual work. Thus, the Helper Server (and ncp_ctrl) must be running so that this method can make the appropriate Telnet request.

Returns

Upon completion, the `GetTelnetCols` method returns the data corresponding to the Telnet command executed during the Telnet session. This data must meet the regular expression supplied in the *$regExp* parameter.

GetTraceRoute

The `GetTraceRoute` method traces a route to the specified destination IP address and returns the network devices that reside on that route.

Method Synopsis

`GetTraceRoute(`*`$ipAddress`*` [, $protocol])`

Parameters

$ipAddress
Specifies the destination IP address whose route is to be traced.

$protocol
Specifies an optional parameter that identifies the IP protocol. You can specify one of the following values:

- `1` — Specifies Internet Protocol version 4 (IPv4).
- `3` — Specifies Internet Protocol version 6 (IPv6).

Description

The `GetTraceRoute` method traces a route to the destination IP address specified in the *ipAddress* parameter. Network devices that exist at the IP addresses on the route will respond to ping requests.

Notes

The `GetTraceRoute` method issues the appropriate ping request to the Helper Server, which performs the actual work. Thus, the Helper Server (and ncp_ctrl) must be running so that this method can make the appropriate ping request.

Returns

Upon completion, the `GetTraceRoute` method returns a list of the devices that reside at IP addresses on the route ending with the destination address specified in the *$ipAddress* parameter.

LockThreads

The `LockThreads` method acquires a lock that only a single agent thread may hold at any given time.

Method Synopsis

`LockThreads()`

Parameters

None

Description

The `LockThreads` method provides a way for a discovery agent to acquire a lock that only a single agent thread may hold at any given time. This means that the code within the locked section is serialized. You should release the lock by calling the `UnLockThreads` method.

Example Usage

The following example shows a call to the `LockThreads` method followed by a call to the `UnLockThreads` method to release the lock. The example assumes a previous call to the `RIV::Agent` constructor, which returns a `RIV::Agent` object (represented by `$agent->`).

```
$agent->LockThreads();

    #
    # Serialised code goes here
    #

    $agent->UnLockThreads();
```

Returns

Upon completion, the `LockThreads` method does not return any values.

PingIP

The `PingIP` method pings the specified IP address.

Method Synopsis

```
PingIP($ipAddress [, $protocol])
```

Parameters

$ipAddress
Specifies the IP address to be pinged.

$protocol
Specifies an optional parameter that identifies the IP protocol. You can specify one of the following values:
- 1 — Specifies Internet Protocol version 4 (IPv4).
- 3 — Specifies Internet Protocol version 6 (IPv6).

Description

The `PingIP` method pings the IP address specified in the *ipAddress* parameter. The method returns without waiting for a response from the network device at that address.

Notes

The `PingIP` method issues the appropriate ping request to the Helper Server, which performs the actual work. Thus, the Helper Server (and ncp_ctrl) must be running so that this method can make the appropriate ping request.

Returns

Upon completion, the `PingIP` method returns the value 1 to indicate that it successfully pinged the device at the specified address. Otherwise, it returns the value 0 (zero).

PingList

The `PingList` method pings the specified list of IP addresses.

Method Synopsis

`PingList(`*`$ipAddressList`*` [, $protocol])`

Parameters

$ipAddressList
Specifies the list of IP addresses to be pinged.

$protocol
Specifies an optional parameter that identifies the IP protocol. You can specify one of the following values:

- `1` — Specifies Internet Protocol version 4 (IPv4).
- `3` — Specifies Internet Protocol version 6 (IPv6).

Description

The `PingList` method pings the list of IP addresses specified in the *ipAddressList* parameter. The method returns without waiting for responses from the network devices at the list of addresses.

Notes

The `PingList` method issues the appropriate ping request to the Helper Server, which performs the actual work. Thus, the Helper Server (and ncp_ctrl) must be running so that this method can make the appropriate ping request.

Returns

Upon completion, the `PingList` method returns the value 1 to indicate that it successfully pinged the devices at the specified addresses. Otherwise, it returns the value 0 (zero).

PingSubnet

The `PingSubnet` method pings the specified subnet.

Method Synopsis

`PingSubnet(`*`$subnet`*`, `*`$netMask`*` [, $protocol])`

Parameters

$subnet
Specifies the IP address of the subnet to be pinged. Typically, subnets are defined as all devices whose IP addresses have the same prefix. Thus, all devices with IP addresses that start with `1.1.1` would be part of the same subnet.

$netmask
Specifies the mask used to determine the subnet to which an IP address belongs.

$protocol
Specifies an optional parameter that identifies the IP protocol. You can specify one of the following values:

- 1 — Specifies Internet Protocol version 4 (IPv4).
- 3 — Specifies Internet Protocol version 6 (IPv6).

Description

The `PingSubnet` method pings the subnet specified in the *subnet* parameter. The method returns without waiting for responses from the network devices that reside on the specified subnet.

Notes

The `PingSubnet` method issues the appropriate ping request to the Helper Server, which performs the actual work. Thus, the Helper Server (and ncp_ctrl) must be running so that this method can make the appropriate ping request.

Returns

Upon completion, the `PingSubnet` method returns the value 1 to indicate that it successfully pinged the devices at the specified subnet. Otherwise, it returns the value 0 (zero).

SendNEToDisco

The `SendNeToDisco` method sends a processed `RIV::Record` to the returns table of the particular Agent database in DISCO.

Method Synopsis

`SendNEToDisco(`*`$entity`*`, `*`$lastRecTag`*`)`

Parameters

$entity
Specifies a reference to a hash list that contains the definition of the record to be sent to DISCO. For convenience, the `RIV::Record` module is such a hash list that provides nested structures for representing local and remote neighbors.

$lastRecTag
Specifies the record for the network entity according to the following values:

- 0 (zero) — Indicates that more records for this network entity are to follow.
- 1 — Indicates the last record for this network.

Note: If you use `RIV::Record` module objects, this parameter is ignored.

Description

The `SendNEToDISCO` method sends a processed *$entity* record object to the returns table of the particular Agent database in DISCO. Typically, the *$entity* parameter is a `RIV::Record` module object that contains information about local and remote neighbors.

Example Usage

```
$TestNE=new RIV::Record($data);
..
..
..
$agent->SendNEToDisco($TestNE,0);
```

Returns

None

SendNEToNextPhase

The `SendNEToNextPhase` method is called by discovery agents that accept data during multiple phases of a network discovery operation. These "multi-phased" discovery agents call `SendNEToNextPhase` when any data processing for a given phase (for example, phase 1) has been completed.

Method Synopsis

`RIV::Agent::SendNEToNextPhase(`*`$entity`*`)`

Parameters

$entity
: Specifies the network entity to be processed and then marked as having been processed for a specific discovery phase.

Description

The `SendNEToNextPhase` method marks the network entity specified in the *$entity* parameter as having completed processing in the current discovery phase, and it puts the network entity back on the Agent queue ready for processing in the next discovery phase.

Each discovery agent maintains an Agent queue that contains network entities sent to it from the DISCO process. A typical discovery agent processes the network entities on its Agent queue and then calls the `SendNEToDisco` method to return the processed network entity to the DISCO process.

Unlike a typical discovery agent, a multi-phased discovery agent must allow for the fact that each discovery phase can be hours apart. Therefore, a multi-phased discovery agent must make multiple calls to the `SendNEToNextPhase` method to put the network entity back on the Agent queue and mark it as ready for processing in the next discovery phase. Once it completes processing of the network entity, the multi-phased discovery agent calls the `SendNEToDisco` method to send the data back to the DISCO process.

The following is the basic flow for a multi-phased discovery agent:
- The DISCO process sends a record (network entity) that provides details about a device that this phased discovery agent can process.
- The multi-phased discovery agent receives this record.
- When free, the multi-phased discovery agent starts processing the record in discovery phase 1. When processing is complete in discovery phase 1, the multi-phased discovery agent calls `SendNEToNextPhase` to put the record back on the Agent queue and mark it as ready for processing in the next discovery phase.

During any of the discovery phases, a multi-phased agent may also be sending multiple data requests to the Helper Server through the `GetSnmp` and `GetTelnet` methods that the `RIV::Agent` module provides.

- When the phase changes, the DISCO process sends out a broadcast indicating that it is proceeding to the next phase (for example, discovery phase 2). When free, the multi-phased discovery agent starts processing the record marked ready for processing in discovery phase 1. When processing is complete in discovery phase 2, the discovery agent calls `SendNEToNextPhase` to put the record back on the Agent queue and mark it as ready for processing in the next discovery phase.
- When the phase changes, the DISCO process sends out a broadcast indicating that it is proceeding to the next phase (for example, discovery phase 3). When free, the multi-phased discovery agent completes processing of the record marked ready for processing in discovery phase 2 and calls `SendNEToDisco` to send all of the data back to the DISCO process.

Notes

You invoke the `SendNEToNextPhase` method on the `RIV::Agent` object returned in a previous call to the `RIV::Agent` constructor. For example:

```
.
.
.
my $agent;
my $agentName = "CiscoSwitchInPerl";
.
.
.
$agent=new RIV::Agent($param, $agentName);
$agent->SendNEToNextPhase($TestNE);
.
.
.
```

Example Usage

The example that illustrates calls to the `SendNEToNextPhase` and `SendNEToDisco` methods is divided into the following sections:

- Create a new multi-phased agent
- Setup for discovery phase-dependent processing
- Setup for discovery phase 1 processing
- Setup for discovery phase 2 processing
- Setup for discovery phase 3 processing

Create a new multi-phased agent

```
.
.
.
my $agent;
my $agentName = "CiscoSwitchInPerl";

sub Init{
    my $param=new RIV::Param();
    $agent=new RIV::Agent($param, $agentName);
}
.
.
.
```

The previous code:

- Declares two variables. The *$agent* variable stores the discovery agent application session object identifier returned by the `RIV::Agent` constructor. The *$agentName* variable stores the name of the agent, which in this example is `CiscoSwitchInPerl`.
- The call to the `RIV::Param` constructor returns an object of type `RIV::Param` to the *$param* variable.
- The call to the `RIV::Agent` constructor takes two parameters: the `RIV::Param` object (*$param*) and the name of the agent (*$agentName*). The `RIV::Agent` constructor returns an agent session object for use in the subsequent call to the `SendNEToNextPhase` method.

Setup for discovery phase-dependent processing

The following code shows the setup for phase-dependent processing:

```
sub ProcessPhase($){
 my $phaseNumber = shift;

 if($RIV::DebugLevel >= 1)
 {
  print "Phase number is $phaseNumber\n";
 }
}
```

Setup for discovery phase 1 processing

The following code shows the setup for phase 1 processing, including the call to the `SendNEToNextPhase` method and calls to the `SnmpGetNext` method. Note that the calls to the `SendNEToNextPhase` and `SnmpGetNext` methods are made through the agent session object (*$agent*) returned in the previous call to the `RIV::Agent` constructor. The `SendNEToNextPhase` method:

- Marks the network entity (*$TestNE*) as having been processed for phase 1.
- Puts this network entity on the `CiscoSwitchInPerl` agent queue ready for phase 2 processing.

```
sub ProcessPhase1($){
 my $TestNE = shift;
.
.
.
BuildVlanData($TestNE);
 my $refVlanIfIndex=$agent->SnmpGetNext($TestNE,'vlanIfIndex');
BuildCardPortToIfIndexData($TestNE);

 my $refLphysAddress=$agent->SnmpGetNext($TestNE,'ifPhysAddress');
.
.
.
$agent->SendNEToNextPhase($TestNE);
}
```

Setup for discovery phase 2 processing

A second call to the `SendNEToNextPhase` method causes the network entity to be marked as having been processed for phase 2 and to be added to the `CiscoSwitchInPerl` agent queue ready for phase 2 processing.

```
sub ProcessPhase2($){
 my $TestNE = shift;
.
.
.
$agent->SendNEToNextPhase($TestNE);
}
```

Setup for discovery phase 3 processing

The following code sets up discovery phase 3 processing: Finally, the multi-phased discovery agenta third call to the `SendNEToNextPhase` method causes the network entity to be marked as having been processed for phase 3 and that it need not be added to the `CiscoSwitchInPerl` agent queue because there is no additional phase processing required. This multi-phased agent also sends back to the DISCO process the `SendNEToNextPhase` record set to the value 1 to indicate the final tokenA second parameter to the `SendNEToNextPhase` method, the value 0 (zero), signifies to the DISCO process that this is the last record token and no additional phase processing is required.

```
sub ProcessPhase3($){
 my $TestNE = shift;
.
.
.
$TestNE->{'m_LastRecord'}=1;
.
.
.
$agent->SendNEToDisco($TestNE,0);
}
```

The `CiscoSwitchInPerl` discovery agent sends back the data associated with this network entity. Because there is no further processing required for this network entity, the `CiscoSwitchInPerl` discovery agent:

- Sets the `m_LastRecord` to the value 1 to indicate the final token and to let the DISCO process know that processing is complete for this network entity.
- Passes the value 0 (zero) as the second parameter in the call to `SendNEToDisco`. A multi-phased agent receives a single network entity, but it may return to the DISCO process several records (one for each local neighbor entry and one for each remote neighbor entry). The DISCO process determines that the multi-phased discovery agent has finished processing a network entity when the value 0 is specified in the call to `SendNEToDisco` to indicate the last record token.

Returns

Upon completion, the `SendNEToNextPhase` method returns no value.

See also

- "RIV::Agent Constructor" on page 73
- "SendNEToDisco" on page 86

SnmpGet

The `SnmpGet` method retrieves the appropriate SNMP information from the Helper Server.

Method Synopsis

`SnmpGet(`*`$ne`*`, `*`$oid`*` [,`*`$instance`*`, `*`$communitySuffix`*`] )`

Parameters

$ne Specifies a reference to the network entity. Typically, this network entity is a `RIV::Record` object.

$oid Specifies a MIB variable (for example, *ifIndex*).

$instance
Specifies the instance of the MIB variable. This is an optional parameter.

$communitySuffix
Specifies the suffix to the community string. This is an optional parameter.

Description

The `SnmpGet` method retrieves the specified SNMP information for the network entity specified in the *$ne* parameter.

Notes

The `SnmpGet` method issues the appropriate SNMP request to the Helper Server, which performs the actual work. Thus, the Helper Server (and ncp_ctrl) must be running so that this method can make the appropriate SNMP request.

Example Usage

```
$varOp = $agent->SnmpGet($NE, 'sysDescr');
print "$varop->{ASN1}", "$varop->{VALUE}", "\n";
```

Returns

Upon completion, the `SnmpGet` method returns a *varop* that contains two key value pairs. The keys are `ANS1` and `value`. The `ANS1` value is the index value after the `OID` corresponding to the MIB variable is removed. It is a single number for MIB variables indexed on a single key and a dot notation for MIB variables indexed by multiple keys.

Note: The `ANS1` value obtained using the `RIV::SnmpAccess` module is the complete `OID` that needs to be split, whereas the `ANS1` value returned by the Helper Server is only the index part.

SnmpGetBulk

The `SnmpGetBulk` method retrieves SNMP GETBULK information from the Helper Server.

Method Synopsis

`SnmpGetBulk(`*`$ne`*`, `*`$oidList`*`, `*`$nonRepeaters`*`,`*`maxRepeaters`*` [,`*`$communitySuffix`*`] )`

Parameters

$ne Specifies a reference to the network entity. Typically, this network entity is a `RIV::Record` object.

$oidList
Specifies a reference to an array of MIB variables. For example:

```
@oids=('sysDescr','sysContact','sysUpTime','ipInReceives',
'ipOutRequests','ipOutDiscards','ipForwDatagrams',
'tcpCurrEstab', 'ifDescr');
$oidList = \@oids;
```

$noRepeaters
Specifies the number of MIB values at the start of the array of MIB variables that return a single value. For example, the `'sysDescr'` MIB variable from the `@oids` array returns a single value.

$maxRepeaters
This parameter is for any MIB variable (for example, `ifIndex`) in the array of MIB variables that returns a table. This parameter specifies the number of values in the table that are to be returned. For example, the value 2 returns only the first two entries. If all the entries are to be returned, *$maxRepeaters* is set to a large number.

$communitySuffix
Specifies the suffix to the community string.

Description

The `SnmpGetBulk` method retrieves SNMP GETBULK information for the network entity specified in the *$ne* parameter.

Notes

The `SnmpGetBulk` method issues the appropriate SNMP request to the Helper Server, which performs the actual work. Thus, the Helper Server (and ncp_ctrl) must be running so that this method can make the appropriate SNMP request.

Example Usage

```
'ipOutRequests','ipOutDiscards','ipForwDatagrams','tcpCurrEstab',
'ifDescr');
($vap) = $agent->SnmpGetBulk($nodeIP, \@oids, 3, 100);
foreach my $varop (@{ $vap})
{
print "$varop->{ASN1}", "$varop->{VALUE}", "\n";
}
```

Returns

Upon completion, the `SnmpGetBulk` method returns a reference to a *varop* array. Each *varop* array contains two key value pairs. The keys are `ANS1` and `VALUE`. The `ANS1` value is the index value after the `OID` corresponding to the MIB variable is removed. It is a single number for MIB variables indexed on a single key and a dot notation for MIB variables indexed by multiple keys.

Note: The ANS1 value obtained using the RIV::SnmpAccess module is the complete OID that needs to be split, whereas the ANS1 value returned by the Helper Server is only the index part.

SnmpGetNext

The SnmpGetNext method retrieves the appropriate SNMP information from the Helper Server.

Method Synopsis

SnmpGetNext(*$ne*, *$oid* [,*$instance*, *$communitySuffix*])

Parameters

$ne Specifies a reference to the network entity. Typically, this network entity is a RIV::Record object.

$oid Specifies a MIB variable (for example, *ifIndex*).

$instance
Specifies the instance of the MIB variable. This is an optional parameter.

$communitySuffix
Specifies the suffix to the community string. This is an optional parameter.

Description

The SnmpGetNext method retrieves the specified SNMP information for the network entity specified in the *$ne* parameter. If *$instance* is defined, the MIB sub-tree starting at that particular instance is retrieved. The *$instance* parameter must be specified as an ASN1 string (for example, "5.3.15").

Notes

The SnmpGetNext method issues the appropriate SNMP request to the Helper Server, which performs the actual work. Thus, the Helper Server (and ncp_ctrl) must be running so that this method can make the appropriate SNMP request.

Example Usage

```
$varOpArray = $agent->SnmpGetNext($NE, 'ifDescr');
foreach my $varop (@{ $varOpArray})
{
print "$varop->{ASN1}", "$varop->{VALUE}", "\n";
}
```

Returns

Upon completion, the SnmpGetNext method returns a reference to a *varop* array. Each *varop* array contains two key value pairs. The keys are ANS1 and VALUE. The ANS1 value is the index value after the OID corresponding to the MIB variable is removed. It is a single number for MIB variables indexed on a single key and a dot notation for MIB variables indexed by multiple keys.

Note: The ANS1 value obtained using the RIV::SnmpAccess module is the complete OID that needs to be split, whereas the ANS1 value returned by the Helper Server is only the index part.

UnLockThreads

The `UnLockThreads` method releases the lock previously acquired in a call to the `LockThreads` method.

Method Synopsis

```
UnLockThreads()
```

Parameters

None

Description

The `UnLockThreads` method releases the lock previously acquired in a call to the `LockThreads` method.

Example Usage

The following example shows a call to the `LockThreads` method followed by a call to the `UnLockThreads` method to release the lock. The example assumes a previous call to the `RIV::Agent` constructor, which returns a `RIV::Agent` object (represented by `$agent->`).

```
$agent->LockThreads();

    #
    # Serialised code goes here
    #

    $agent->UnLockThreads();
```

Returns

Upon completion, the `UnLockThreads` method does not return any values.

RIV::App module reference

The `RIV::App` module provides an interface for implementing Network Manager client/server applications within one domain.

The `RIV::App` module provides two constructors that instantiate a `RIV::App` object. The constructors are described in reference (man) page format.

Note: The `RIV::App` module does not provide any methods or functions.

RIV::App module synopsis

The `RIV::App` module synopsis shows how to make calls to the two constructors that this module provides.

The comments provided in the synopsis serve as a quick reference as to the purpose of the constructors. The reference (man) page for the constructors provides the details.

```
# Load the RIV::App module.
use Riv::App;
#
   # Call the first form of the RIV::App constructor, passing to $domain the
   # name of the Network Manager domain. The constructor returns
   # a RIV::App object to $rivApp.
$rivApp = new RIV::App($domain, $progname, $doHeartBeat);
#
   # Call the second form of the RIV::App constructor, passing as the
   # first parameter a RIV::Param object that was returned
   # in a previous call to the RIV::Param constructor. The constructor
   # returns a RIV::App object to $rivApp.
$rivApp = new RIV::App(RIV::Param, $progname, $doHeartBeat);
```

RIV::App Constructor

The `RIV::App` constructor creates and initializes a new application session.

Constructor

`new(`*`$domain`*`, `*`$progname`*` [, `*`$doHeartBeat`*`])`

`new(RIV::Param, `*`$progname`*` [, `*`$doHeartBeat`*`])`

Parameters

$domain
: Specifies a Network Manager IP Edition domain name.

  Note: A default domain name is not supported.

RIV::Param
: Specifies a `RIV::Param` object that was returned in a previous call to the `RIV::Param` constructor. The `RIV::Param` object contains a parsed form of the command line arguments.

$progname
: Specifies a parameter used when building fault-tolerant server groups. It must contain a string that uniquely identifies the application. By convention, the application name should start with `ncp_`.

$doHeartBeat
: Specifies an optional parameter that indicates whether the application generates a heartbeat signal. If the application generates a heartbeat signal, set this parameter to a nonzero value.

Description

The `RIV::App` constructors create and initialize a new application session. The constructors differ in that one takes a *$domain* parameter and the other takes a `RIV::Param` parameter.

Example Usage

```
$app = RIV::App::new("foo", "ncp_test", 1);

#!$NCHOME/bin/ncp_perl
use RIV;
use RIV::App;
my $app = RIV::App::new("MYDOMAIN", "ncp_test");
...
undef $app;
```

Returns

Upon completion, the `RIV::App` constructors return a `RIV::App` object that encapsulates the new application session.

See Also

- "RIV::Param Constructor" on page 106

RIV::OQL module reference

The RIV::OQL module provides an interface to communicate with and perform operations on Network Manager internal databases.

The RIV::OQL module provides a constructor that allows you to create a new RIV::OQL session object and within this session object call methods to:

- Connect to a particular service type
- Create new databases and tables
- Query the internal databases
- Insert records into and delete records from the internal databases
- Print and update records that reside in the internal databases

The constructor and methods are described in reference (man) page format.

RIV::OQL module synopsis

The RIV::OQL module synopsis shows how to make calls to the constructor and database operation methods that this module provides.

The comments provided in the synopsis serve as a quick reference as to the purpose of the constructor and database operation methods. The reference (man) pages for the constructor and each method provide the details.

```
# Load the RIV::OQL module
use RIV::OQL;

# Call the RIV::OQL constructor, passing to $appSession one of the
# following blessed references:
#
#   + A RIV::Agent object (returned in a previous call to the
#     RIV::Agent constructor)
#   + A RIV::App object (returned in a previous call to the
#     RIV::App constructor)
#
# The $precisionService parameter takes one of the valid Network Manager
# service names, for example, ncp_disco (Disco service).
#
# The calls to the database operation methods are made through a
# reference to the RIV::OQL session object ($oql->) that the RIV::OQL
# constructor returns.
#
$oql = new RIV::OQL($appSession, $precision_Service);

# Call the Send method to send an OQL query to the specified database.
#
$oql->Send($oqlStatement, $returnResults);
#
# Call the CreateDb method to create a database in the Network Manager
# service specified in a previous call to the RIV::OQL constructor.
#
$oql->CreateDB($databaseName);
#
# Call the CreateTable method to create a table in the database.
#
$oql->CreateTable($databaseName, $tableName, \%columnNamesandTypes);
#
# Call the Insert method to insert records into a database table.
```

```
$oql->Insert($database, $table, \%record);
#
# Call the Select method to execute a specific OQL command.
oql->Select($database, $table, $columnName);
#
# Call the RIV module's GetResult function to get input data.
my ($type, $data) = $oql->RIV::GetResult(10);
#
# Call the Print method to print the contents of the records obtained
# as a result of this database query.
$oql->Print($data);
#
# Call the Delete method to delete records from the database table.
$oql->Delete($database, $table, $clauseForDeletion);
#
# Call the Update method to update records that currently reside in
# the database.
$oql->Update($database, $table, $setClause, $whereClause);
```

RIV::OQL Constructor

The `RIV::OQL` constructor creates and initializes a new `RIV::OQL` object.

Constructor

`new($rivSession, $rivService)`

Parameters

$rivSession
Specifies a blessed reference to either a `RIV::App` or `RIV::Agent` object.

$rivService
Specifies the name of a service to indicate the internal database to which this OQL session interacts. The following table identifies the available services to which to connect along with their corresponding executable. For example, the service name Disco indicates that the OQL session will interact with the DISCO databases that the `ncp_disco` executable creates.

Service Name	Executable
Model	ncp_model
Amos	ncp_event
Monitor	ncp_monitor
Class	ncp_class
Store	ncp_store
Exec	ncp_exec
Ctrl	ncp_ctrl
Helper	ncp_d_helpserv
Disco	ncp_disco

Description

The `RIV::OQL` constructor creates and initializes a new `RIV::OQL` session object.

Example Usage

```
$app = new RIV::App();
$oql = new RIV::OQL($app, 'Disco');
```

Returns

Upon completion, the `RIV::OQL` constructor returns a `RIV::OQL` session object.

See Also

- "RIV::Agent Constructor" on page 73
- "RIV::App Constructor" on page 95

CreateDB

The `CreateDB` method creates a database.

Method Synopsis

`CreateDB(`*`$databaseName`*`)`

Parameters

$databaseName
Specifies the name of the database to be created.

Description

The `CreateDB` method creates a database with the name *$databaseName* in the specified service. You specified this service in the *$rivService* parameter in a previous call to the `RIV::OQL` constructor.

Example Usage

The following example shows how to create a new database, with the name `foo`, in the Disco service for which an OQL session was created using the `RIV::OQL` constructor.

```
$oql = new RIV::OQL($app, 'Disco');
$oql->CreateDB("foo");
```

Returns

Upon completion, the `CreateDB` method does not return any records.

See Also

- "RIV::OQL Constructor" on page 97"RIV::Agent Constructor" on page 73

CreateTable

The `CreateTable` method creates a database table.

Method Synopsis

`CreateTable(`*`$databaseName`*`, `*`$tableName`*`, \`*`%columnNames`*`)`

Parameters

$databaseName
Specifies the name of the database in which the table is to be created.

$tableName
Specifies the name of the table to be created.

\%columnNames
: Specifies a hash list of the columns in the table. The keys in the hash list refer to the column name and the values are one of the types supported by the OQL syntax.

Description

The `CreateTable` method creates a database table with the name specified in the *$tableName* parameter in the database specified in the *$databaseName* parameter.

You created this database in previous calls to the:
- `RIV::OQL` constructor — You specified the name of a service (in the *$rivService* parameter) to indicate the internal database to which this `RIV::OQL` session object interacts.
- `CreateDB` method — You specified the name of the database (in the *$databaseName* parameter) to be created in the service specified in the call to the `RIV::OQL` constructor.

Example Usage

The following example shows how to create:
- A new database, with the name `foo`, in the Disco service for which a `RIV::OQL` session object was created in a call to the `RIV::OQL` constructor.
- Column names (`m_IpAddress` and `m_Name`) and associated values to appear in the table.
- A table called `bar`.

```
$oql = new RIV::OQL($app, 'Disco');
$oql->CreateDB("foo");
%columnNames = ("m_IpAddress"=> "text", "m_Name"=> "text");
$oql->CreateTable("foo", "bar", \%columnNames);
```

Returns

Upon completion, the `CreateTable` method does not return any records.

See Also
- "RIV::OQL Constructor" on page 97
- "CreateDB" on page 98

Delete

The `Delete` method deletes records from a database table.

Method Synopsis

`Delete(`*`$databaseName`*`, `*`$tableName`*`, `*`$clause`*`)`

Parameters

$databaseName
: Specifies the name of the database from which records are to be deleted.

$tableName
: Specifies the name of the table in the specified database (*$databaseName*) from which records are to be deleted.

$clause
Specifies any valid OQL comparative statement used as a condition for deleting records. If a record matches *$clause*, the `Delete` method will delete it.

Description

The `Delete` method deletes records from the table specified in the *$tableName* parameter that resides in the database specified in *$databaseName* parameter and that satisfy the criteria defined by the OQL comparative statement specified in the *$clause* parameter.

You created this database and database table in previous calls to the:

- `RIV::OQL` constructor — You specified the name of a service (in the *$rivService* parameter) to indicate the internal database to which this `RIV::OQL` session object interacts.
- `CreateDB` method — You specified the name of the database (in the *$databaseName* parameter) to be created in the service specified in the call to the `RIV::OQL` constructor.
- `CreateTable` method — You specified the name of the database table (in the *$tableName* parameter) to be created in the database specified in the call to the `CreateDB` method.

Example Usage

The following example shows how to delete records from:

- A database called `master`.
- A table called `entityByName`.

The records to be deleted are those with an `EntityOID` equal to the value `1.3.6.1.4.1.42.2.1.1`.

```
$oql->Delete('master', 'entityByName', "EntityOID = '1.3.6.1.4.1.42.2.1.1'");
```

Returns

Upon completion, the `Delete` method does not return any records.

See Also

- “RIV::OQL Constructor” on page 97
- “CreateDB” on page 98
- “CreateTable” on page 98

Insert

The `Insert` method inserts records into a database table.

Method Synopsis

`Insert(`*`$databaseName`*`, `*`$tableName`*`, \`*`%record`*`)`

Parameters

$databaseName
Specifies the name of the database in which the record is to be inserted.

$tableName
Specifies the name of the table in the specified database (*$databaseName*) in which the record is to be inserted.

\%record
Specifies a hash list that defines the record to be inserted.

Description

The `Insert` method creates an OQL statement that inserts the record defined by the hash list specified in the *\%record* parameter into the database table specified in the *$tableName* parameter that resides in the database specified in the *$databaseName* parameter.

You created this database and database table in previous calls to the:

- `RIV::OQL` constructor — You specified the name of a service (in the *$rivService* parameter) to indicate the internal database to which this `RIV::OQL` session object interacts.
- `CreateDB` method — You specified the name of the database (in the *$databaseName* parameter) to be created in the service specified in the call to the `RIV::OQL` constructor.
- `CreateTable` method — You specified the name of the database table (in the *$tableName* parameter) to be created in the database specified in the call to the `CreateDB` method.

Example Usage

The following example shows how to insert the record specified in the *\%record* parameter in:

- A database called `finders`.
- A table called `despatch`.

Note:

The service used to create the `RIV::OQL` object (`$oql->`) in a previous call to the `RIV::OQL` constructor has to be `Disco`.

```
%record = ( m_Creator => 'PerlDetails',
m_Name => 'foo',
m_IpAddress => '123.1.2.3',);
$oql->Insert('finders', 'despatch', \%record);
```

Returns

Upon completion, the `Insert` method does not return any records.

See Also

- "RIV::OQL Constructor" on page 97
- "CreateDB" on page 98
- "CreateTable" on page 98

Print

The `Print` method prints records obtained as a result of a database query.

Method Synopsis

`Print(`*`$data`*`)`

Parameters

$data Specifies a reference to an array of hash lists that represent the records obtained from the `SELECT` statement.

Description

The `Print` method prints the records obtained as a result of a query.

Example Usage

The following example shows how to:

- Use the `Select` method to execute a `SELECT` statement.
- Use the `RIV::GetResult` method to specify the number of seconds to wait (in the example, 10 seconds) for input before returning.
- Print the data specified in *$data*.

```
$oql->Select('class','activeClasses', 'ALL');
my ($type, $data) = $oql->RIV::GetResult(10);
Print ($data);
```

Returns

Upon completion, the `Print` method does not return any records.

See Also

- "RIV::GetResult" on page 63

Select

The `Select` method executes a specific OQL statement.

Method Synopsis

`Select(`*`$databaseName`*`, `*`$tableName`*`, `*`$columnName`*`)`

Parameters

$databaseName
Specifies the name of the database in which the OQL statement is to be executed.

$tableName
Specifies the name of the table in the specified database (*$databaseName*) in which the OQL statement is to be executed.

$columnName
Specifies the name of the column for which the results are to be returned. If all entries are to be returned, set the *$columnName* parameter to `ALL`.

Description

The `Select` method executes the following OQL statement:

```
select $columnName from $dbName.$tableName;
```

When the *$columnName* parameter is set to `ALL`, the `Select` method executes the following OQL statement:

```
select * from $dbName.$tableName;
```

You created this database and database table in previous calls to the:

- `RIV::OQL` constructor — You specified the name of a service (in the *$rivService* parameter) to indicate the internal database to which this `RIV::OQL` session object interacts.
- `CreateDB` method — You specified the name of the database (in the *$databaseName* parameter) to be created in the service specified in the call to the `RIV::OQL` constructor.
- `CreateTable` method — You specified the name of the database table (in the *$tableName* parameter) to be created in the database specified in the call to the `CreateDB` method.

Example Usage

```
$statement = "select * from master.entityByName;";
$oql->Send($statement, 1);
my ($type, $data) = $oql->RIV::GetResult(10);
```

The results are obtained by using the `RIV::GetResult` method. For example:

```
$oql->Select('class', 'activeClasses', 'ALL');
my ($type, $data) = $oql->RIV::GetResult(10);
```

In the previous example:

- The *$type* parameter specifies the tag `OQLQuery`.
- The *$data* parameter specifies a reference to an array of hash lists that represents the records obtained from the OQL database query.
- All records are received from the database table called `activeClasses` that resides in the database called `class`. In this case, the service to which this OQL session is connected must be `Class`.

Returns

The results are obtained by using the `RIV::GetResult` method.

See Also

- "RIV::GetResult" on page 63
- "RIV::OQL Constructor" on page 97
- "CreateDB" on page 98
- "CreateTable" on page 98

Send

The `Send` method provides a way to communicate with the databases.

Method Synopsis

`Send(`*`$statement`*`, `*`$returnResults`*`)`

Parameters

$statement
Specifies any valid OQL statement.

$returnResults
Specifies whether to return results. This parameter takes one of the following values:

- 1 – Specify the value 1 for database queries (for example, OQL statements such as `select` and `show`) that return results.
- 0 – Specify the value 0 (zero) for database queries (for example, OQL statements such as `insert`, `update`, and `delete`) that do not return results.

Description

The `Send` method provides a way to communicate with the databases. The *$statement* parameter specifies any valid OQL statement that the `Send` method executes. The *$returnResults* parameter indicates whether you are interested in the results of the OQL statement. For example, when an OQL `select` statement is executed and you are interested in the results, the *$returnResults* parameter must be set to the value 1. The `RIV::GetResult` method is used to receive the results.

Example Usage

```
$statement = "select * from master.entityByName;";
$oql->Send($statement, 1);
my ($type, $data) = $oql->RIV::GetResult(10);
```

Returns

Upon completion, the `Send` method returns the results of the OQL statement. If you set *$returnResults* to 0 (zero), the `Send` method does not return any records.

See Also

- "RIV::GetResult" on page 63
- "Select" on page 102

Update

The `Update` method updates records that currently reside in the database.

Method Synopsis

`Update(`*`$databaseName`*`, `*`$tableName`*`, `*`$setClause`*`, `*`$whereClause`*`)`

Parameters

$databaseName
Specifies the name of the database in which the record is to be updated.

$tableName
: Specifies the name of the table in the specified database (*$databaseName*) in which the record is to be updated.

$setClause
: Specifies the clause that defines `set the variable to`.

$whereClause
: Specifies the clause that defines `where the variable is`.

Description

The `Update` method updates records that already reside in the database and database table specified in the *$databaseName* and *$tableName* parameters, respectively. Calling the `Update` method is equivalent to executing the following OQL statement:

```
UPDATE $databaseName.$tableName SET $setClause WHERE $whereClause;
```

You created this database and database table in previous calls to the:

- `RIV::OQL` constructor — You specified the name of a service (in the *$rivService* parameter) to indicate the internal database to which this `RIV::OQL` session object interacts.
- `CreateDB` method — You specified the name of the database (in the *$databaseName* parameter) to be created in the service specified in the call to the `RIV::OQL` constructor.
- `CreateTable` method — You specified the name of the database table (in the *$tableName* parameter) to be created in the database specified in the call to the `CreateDB` method.

Example Usage

The following example does the following:

- Updates specific records in the table called `entityByName` that resides in the database called `master`.
- The specific records updated are those that have `EntityName foo` to `EntityName ppp`.

```
$oql->Update('master', 'entityByName', "EntityName='ppp'",
"EntityName='foo'");
```

Returns

Upon completion, the `Update` method does not return any records.

See Also

- "RIV::OQL Constructor" on page 97
- "CreateDB" on page 98
- "CreateTable" on page 98

RIV::Param module reference

The `RIV::Param` module provides an interface for parsing standard and Network Manager application-specific command line arguments.

The `RIV::Param` module provides a constructor that creates a new `RIV::Param` object that you use to call methods that perform the following tasks:

- Obtain the name of a command
- Obtain the name of a domain
- Print a brief usage explanation to standard output

The constructor and methods are described in reference (man) page format.

RIV::Param module synopsis

The `RIV::Param` module synopsis shows how to make calls to the constructor and parameter operation methods that this module provides.

The comments provided in the synopsis serve as a quick reference as to the purpose of the constructor and parameter operation methods. The reference (man) pages for the constructor and each method provide the details.

```
# Load the RIV::Param module
use RIV::Param;
#
# These are the RIV::Param module constants used to specify
# whether a command line parameter takes no arguments or a
# single argument and whether it is mandatory.
RivParamNoArg, RivParamSingleArg;
RivParamMandatory;
#
# Call the RIV::Param constructor. The RIV::Param constructor
# returns to $param a new RIV::Param object.
#
$param = RIV::Param::new(\%paramHash, \@usageStrings, \$helpMessage);
#
# Use the RIV::Param object ($param->) to invoke the methods
# that the RIV::Param module provides.

# Call the Usage method to print a brief usage explanation to
# standard output.
$param->Usage($errorCode);
#
# Call the DomainName method to obtain the name of the domain.
$domainName = $param->DomainName();
#
# Call the CommandName method to obtain the name of the command.
$commandName = $param->CommandName();
```

RIV::Param Constructor

The `RIV::Param` constructor creates and initializes a new `RIV::Param` object.

Constructor

```
$param = RIV::Param::new([\%paramHash,\@usageStrings, \$helpMessage])
```

Parameters

\%paramHash
Specifies a reference to a hash used to specify application-specific

command line arguments. Each hash key (index) represents a command line switch and its associated hash value is an array with the following elements:

- element 0 — Is a `flags` element that specifies a bitwise `OR` that indicates:
  - Whether the command line switch takes an argument.
  - Whether the switch is mandatory

  The `flags` element makes use of the package constants described in "Package Constants."
- element 1 — Is a scalar variable reference or `undef` value. You initialize the scalar variable reference with the appropriate value (a parameter from the command line or 1).

The \%*paramHash* parameter is optional.

\@usageStrings
Specifies either a string that contains usage information or an array reference that contains an element for each of the nonstandard command line argument scenarios. If the application takes only standard arguments, this constructor argument (if specified) should be set to the `undef` value.

The \@*usageStrings* parameter is optional.

\$helpMessage
Specifies a string reference that contains explanatory information that is written to standard output, in addition to standard help information, when the `-help` command line argument is specified.

The \$*helpMessage* parameter is optional.

Package Constants

The `RIV::Param` constructor's \%*paramHash* parameter makes use of the following package constants:

- `RivParamNoArg` — Specifies that the command line parameter takes no arguments.
- `RivParamSingleArg` — Specifies that the command line parameter takes one argument.
- `RivParamMandatory` — Specifies that the command line parameter is mandatory, and that it is a fatal error for the parameter to be missing.

Description

The `RIV::Param` constructor creates and initializes a new `RIV::Param` object from the application-specific command line arguments specified in the \%*paramHash* parameter. Each new `RIV::Param` object also encapsulates the supported standard command line arguments. Thus, Network Manager client/server and Agent applications can make use of these standard command line arguments in addition to the application-specific command line arguments.

If you call the `RIV::Param` constructor without specifying any of the optional parameters, the new `RIV::Param` object provides access to the standard command line arguments.

Example Usage No Parameters

The following code shows a call to the `RIV::Param` constructor without specifying any of the optional parameters. The newly created `RIV::Param` object is then passed to the `RIV::Agent` constructor, which returns a `RIV::Agent` object that provides a discovery agent application session. In this example, the discovery agent called `PerlDetails` (the name specified in the second parameter of the `RIV::Agent` constructor) can make use of the standard command line arguments.

```
.
.
.
sub Init{
    my $param=RIV::Param::new();
    $agent=RIV::Agent::new($param,"PerlDetails");
}
.
.
.
```

Example Usage Three Parameters

The example described in this section shows a call to the `RIV::Param` constructor that specifies the three optional parameters.

The following code defines a reference to a hash called `%CmdLineArgs` used to specify application-specific command line arguments. The `%CmdLineArgs` hash is passed as the first parameter to the `RIV::Param` constructor:

```
.
.
.
my $subject;
my $process = 'Model';
my $messageClass = 'NOTIFY';
my $verbose;
my %CmdLineArgs = (
    "-subject"  => [ RivParamSingleArg , \$subject ],
    "-process"  => [ RivParamSingleArg , \$process ],
    "-messageClass"  => [ RivParamSingleArg , \$messageClass ],
    "-verbose"  => [ RivParamNoArg,     \$verbose ]
);
.
.
.
```

The following list provides brief descriptions of the application-specific command line arguments defined in the `%CmdLineArgs` hash.

- `-subject` — Specifies a command line argument that takes one argument (as indicated by the `RivParamSingleArg` package constant). This command line argument also specifies a reference to a scalar value, `\$subject`. It is expected that a user would supply a specific subject on the command line.
- `-process` — Specifies a command line argument that takes one argument (as indicated by the `RivParamSingleArg` package constant). This command line argument also specifies a reference to a scalar value, `\$process`. It is expected that a user would supply the specific process that is of interest (for example, `Class`, `Config`, `Event`, and so forth) on the command line. The default process is `Model`.
- `-messageClass` — Specifies a command line argument that takes one argument (as indicated by the `RivParamSingleArg` package constant). This command line argument also specifies a reference to a scalar value, `\$messageClass`. It is

expected that a user would supply the class of messages that are of interest (for example, QUERY, STATUS, and so forth) on the command line. The default message class is NOTIFY.

- -verbose — Specifies a command line argument that takes no arguments (as indicated by the RivParamNoArg package constant). This command line argument also specifies a reference to a scalar value, \$verbose. It is expected that a user would specify this command line argument to explicitly print out details of nested fields.

The following code defines a usage string called @Usage that provides information on how to use the command line for this application. The @Usage array is passed as the second parameter to the RIV::Param constructor:

```
.
.
.
my @Usage = (
    "[-subject <subject> -process [Model|Disco|Ctrl|...]
     -messageClass [NOTIFY|QUERY|...] "
);
.
.
.
```

The following code defines a string reference called $helpData that contains explanatory information about the application-specific command line arguments and other pertinent information. The explanatory information also includes descriptions of the standard command line arguments (-domain, -debug, and -help). The $helpData string reference is passed as the third parameter to the RIV::Param constructor:

```
my $helpData = "\n
The ITNMIP_Listener perl script is intended to listen on the supplied subject
and print out the messages received.

The arguments are
-domain <domain> = Name of the domain to retrieve data from
-debug [0-4] = Required debug level
-help = This information
-verbose = Explicitly print out details of nested fields
-subject = The specific subject to listen to ( this will not include the domain )
-process = The process to listen to ( e.g. Model, Class, Event, Config, Ctrl ,
           Disco , PingFinder )
-messageClass = The class of messages of interest. Not all processes support
all classes. The common ones of interest are NOTIFY, QUERY, STATUS

The most common arguments to use are
-process Model -messageClass NOTIFY : ( default ) - Listen for the models
 updates on topology changes (old style)
-process Model -messageClass TOPOLOGY : ( default ) - Listen for the models
 updates on topology changes (new style)
-process Disco -messageClass STATUS : - Listen to disco broadcasts on the
 current state of the discovery.
-process DNCIM2NCIM -messageClase NOTIFY : - Listen to Disco to Model
 DNCIM2NCIM updates.
-process ITNMSTATUS -messageClass NOTIFY : - Listen to ITNM status events.

The process is capable of listening on any subject on the message broker bus
but will not decode the output beyond printing out the contents of the message.

The syntax for message broker subjects is

    /<subject>/<sub-subject>/<sub-sub-subject>/....
```

```
All ITNM IP subjects begin \'ITNM/\' and have the domain appended so the
model notify subject for domain TESTDOMAIN is

    /ITNM/MODEL/NOTIFY/TESTDOMAIN



\n";
```

The following code shows the call to the `RIV::Param` constructor using the three previously defined parameters: `\%CmdLineArgs`, `\@Usage`, and `\$helpData`. Note the call to the `die` function to exit the script if the `RIV::Param` constructor fails to create a new `RIV::Param` object.

```
.
.
.
my $param = RIV::Param::new(\%CmdLineArgs, \@Usage, \$helpData);
die "Can't create RIV::Param" unless (defined $param);
.
.
.
```

Returns

Upon completion, the `RIV::Param` constructor returns a new `RIV::Param` object.

See Also

- "RIV::Agent Constructor" on page 73
- "RIV::App Constructor" on page 95
- "Usage" on page 112
- "RIV::Param module overview" on page 8

CommandName

The `CommandName` method returns the name of the specified command.

Method Synopsis

```
RIV::Param::CommandName()
```

Parameters

None

Description

The `CommandName` method returns the name of the command specified on the command line.

Use the `RIV::Param` object returned in a previous call to the `RIV::Param` constructor to invoke the `CommandName` method. For example: `$param->CommandName`.

Example Usage

The following code shows a call to the `CommandName` method. Note that the `CommandName` method is invoked through the newly created `RIV::Param` object returned to the *$param* variable.

```
.
.
.
my $param = RIV::Param::new(\%cmdLineArgs, \@Usage, \$helpData);
die "Can't create RIV::Param" unless (defined $param);
.
.
.
  my $command = $param->CommandName();
.
.
.
```

Returns

Upon completion, the CommandName method returns the name of the command specified on the command line.

See Also

- "RIV::Param Constructor" on page 106

DomainName

The DomainName method returns the name of the specified domain.

Method Synopsis

```
RIV::Param::DomainName()
```

Parameters

None

Description

The DomainName method returns the name of the domain specified on the command line for the -domain standard command line argument.

Use the RIV::Param object returned in a previous call to the RIV::Param constructor to invoke the DomainName method. For example: $param->DomainName.

Example Usage

The following code shows a call to the DomainName method. Note that the DomainName method is invoked through the newly created RIV::Param object returned to the *$param* variable.

```
.
.
.
my $param = RIV::Param::new(\%cmdLineArgs, \@Usage, \$helpData);
die "Can't create RIV::Param" unless (defined $param);
.
.
.
die "ncp_disco must be running under domain ",
        $param->DomainName(),
        " - unable to query the disco.config table"
        unless $dbData;
```

```
    print "...disco is running under domain ", $param->DomainName(), "\n" if $debug;
.
.
.
```

Returns

Upon completion, the DomainName method returns the name of the domain specified on the command line for the -domain standard command line argument.

See Also

- "RIV::Param Constructor" on page 106

Usage

The Usage method writes a brief usage explanation to standard output.

Method Synopsis

RIV::Param::Usage(*$errorCode*)

Parameters

$errorCode
Specifies either a status or the undef value. The status gets written to standard output.

Description

The Usage method writes a brief usage explanation to standard output and then exits with the status specified in the *$errorCode* parameter, if defined. If you specified the undef value in the *$errorCode* parameter, the Usage method returns to the caller.

Use the RIV::Param object returned in a previous call to the RIV::Param constructor to invoke the Usage method. For example: $param->Usage.

Example Usage

The following code shows a call to the Usage method. In this example, an error code of 1 is passed. Note that the Usage method is invoked through the newly created RIV::Param object returned to the *$param* variable.

```
my @_Usage = (    # usage string suffixes
 "<node> [ async ]"
);

#
# Read and parse the command line, standard args are hidden
#
my $param = RIV::Param::new({
  "-v"        => [ $RIV::Param::NoArg, \$Verbose ],
 }, \@_Usage);
die "RIV::Param::new failed" unless defined $param;

my $node        = shift @ARGV;
my $what        = shift @ARGV;
$what = "" unless defined $what;

$param->Usage(1)
 unless (defined $node && $node ne "");
```

Returns

Upon completion, the `Usage` method writes a brief usage message to standard output and the status specified in the *$errorCode* parameter and simply exits. If the *$errorCode* parameter is set to the `undef` value, the `Usage` method returns to the caller.

See Also

- "RIV::Param Constructor" on page 106

RIV::Record module reference

The `RIV::Record` module provides a data structure to store the network entity.

The `RIV::Record` module provides a constructor that creates and initializes a `RIV::Record` data structure. This module also provides methods to perform the following operations:

- Add local neighbors
- Add remote neighbors
- Get local neighbors
- Get remote neighbors
- Print the current records

The constructor and methods are described in reference (man) page format.

RIV::Record module synopsis

The `RIV::Record` synopsis shows how to make calls to the constructor and local and remote neighbors operation methods that this module provides.

```
use RIV::Agent;
use RIV::Record;
my($tag, $data) = $agent->RIV::GetResult(-1);
if($tag eq 'NE'){
foreach $key (@$data){
$NE = RIV::Record::new($key);
}
}
$NE->AddLocalNeighbour($refLocalNeighbour);
$NE->AddRemoteNeighbour($refLocalNeighbour, $refRemoteNeighbour);
$arrayVarOps = $agent->SnmpGetNext($NE, $mibVariable);
$NE->AddLocalNeighbourTag($tagName, $arrayVarOps);
$NE->AddRemoteNeighbourTag($reflocalNeighbour, $tagName, $arrayVarOps);
@localNeighbours = $NE->GetLocalNeighbours();
@remoteNeighbours = $NE->GetRemoteNeighbours($refLocalNeighbour);
$NE->Print();
```

RIV::Record Constructor

The `RIV::Record` constructor creates and initializes a new `RIV::Record` object.

Constructor

`new(`*`$refNE`*`)`

Parameters

$refNE

Specifies a reference to a hash list. The hash list is the mechanism used to store network entity records retrieved from the discovery engine, DISCO.

Description

The `RIV::Record` constructor creates and initializes a new `RIV::Record` object. This object stores network entity records retrieved from DISCO.

Example Usage

The following code fragment illustrates a typical loop for receiving records from DISCO:

```
while (1){
my($tag, $data) = $agent->RIV::GetResult(-1);
# Get the network entities
print "TAG :", $tag, "\n";
if($tag eq 'NE'){
foreach $key (@$data){
$ne = new RIV::Record($key);
}
}
}
```

Returns

Upon completion, the `RIV::Record` constructor returns a `RIV::Record` object.

See Also

- "RIV::Agent Constructor" on page 73
- "RIV::GetResult" on page 63

AddLocalNeighbour

The `AddLocalNeighbour` method adds a local neighbor.

Method Synopsis

`AddLocalNeighbour(`*`$refNbr`*`)`

Parameters

$refNbr
Specifies a reference to a hash list that defines the local neighbor using a set of key value pairs (`varBinds`).

Description

The `AddLocalNeighbour` method adds a local neighbor whose hash list reference is *$refNbr*.

Example Usage

```
$localNbr{'m_IpAddress'} = '1.2.3.4';
$localNbr{'m_IfIndex'} = 2;
$NE->AddLocalNeighbour(\%localNbr);
```

Returns

Upon completion, the `AddLocalNeighbour` method does not return any records.

AddLocalNeighbourTag

The AddLocalNeighbourTag method adds a tag (varBind) to a local neighbor.

Method Synopsis

AddLocalNeighbourTag(*$tag*, *$refVarOp*)

Parameters

$tag Specifies the key value for the varBind.

$refVarOp
Specifies a reference to an array of varops.

Description

The AddLocalNeighbourTag method adds to local neighbors a varBind whose key is defined by the *$tag* parameter and the value defined by the *$refVarOp* parameter (a reference to an array of varops). The key and value are added sequentially, that is, the values in the @$refVarOp array are assumed to be in the same order as the local neighbor array. If local neighbors do not exist, then AddLocalNeighbourTag creates them.

Example Usage

```
$refLifindex=$agent->SnmpGetNext($TestNE, 'ipAdEntIfIndex');
$TestNE->AddLocalNeighbourTag("m_IfIndex", $refLifIndex);
```

Returns

Upon completion, the AddLocalNeighbourTag method does not return any records.

See Also

- "RIV::Agent Constructor" on page 73
- "SnmpGetNext" on page 93

AddRemoteNeighbour

The AddRemoteNeighbour method adds a remote neighbor.

Method Synopsis

AddRemoteNeighbour(*$refLocalNbr*, *$refRemoteNbr*)

Parameters

$refLocalNbr
Specifies a reference to the hash list that defines a local neighbor and to which list the remote neighbor is to be added.

$refRemoteNbr
Specifies a reference to a hash list that defines the remote neighbor using a set of key value pairs (varBinds).

Description

The AddRemoteNeighbour method adds a remote neighbor whose hash list reference is *$refRemoteNbr* to the local neighbor whose hash list reference is *$refLocalNbr*.

Example Usage

```
$remoteNbr{'m_IpAddress'} = '1.2.5.6';
$NE->AddRemoteNeighbour($localNbr, \%remoteNbr);
```

Returns

Upon completion, the `AddRemoteNeighbour` method does not return any records.

AddRemoteNeighbourTag

The `AddRemoteNeighbourTag` method adds a tag (`varBind`) to a remote neighbor.

Method Synopsis

`AddRemoteNeighbourTag(`*`$refLocalNbr`*`, `*`$tag`*`, `*`$refVarOp`*`)`

Parameters

$refLocalNbr
Specifies a reference to the local neighbor to which the remote neighbors are to be added.

$tag Specifies the key value for the `varBind`.

$refVarOp
Specifies a reference to an array of varops.

Description

The `AddRemoteNeighbourTag` method adds to remote neighbors a `varBind` whose key is defined by the *$tag* parameter and the value defined by the *$refVarOp* parameter (a reference to an array of varops). The key and value are added sequentially, that is, the values in the `@$refVarOp` array are assumed to be in the same order as the remote neighbor array. If remote neighbors do not exist, then `AddRemoteNeighbourTag` creates them.

The *$tag* parameter specifies a reference to the local neighbor to which the remote neighbor currently resides or will be added (if it does not currently exist).

Example Usage

```
$refRifIndex = $agent->SnmpGetNext($TestNE,...);
$TestNE->AddRemoteNeighbourTag($refLocal, "m_IfIndex", $refRifIndex);
```

Returns

Upon completion, the `AddRemoteNeighbourTag` method does not return any records.

See Also

- “RIV::Agent Constructor” on page 73
- “SnmpGetNext” on page 93

GetLocalNeighbours

The `GetLocalNeighbours` method returns an array of local neighbors.

Method Synopsis

```
GetLocalNeighbours()
```

Parameters

None

Description

The `GetLocalNeighbours` method returns an array of local neighbors.

Example Usage

```
@localNeighbours = $NE->GetLocalNeighbours();
```

Returns

Upon completion, the `GetLocalNeighbours` method returns an array of local neighbors (as a reference to a hash list).

GetRemoteNeighbours

The `GetRemoteNeighbours` method returns an array of remote neighbors.

Method Synopsis

```
GetRemoteNeighbours($refLocalNeighbour)
```

Parameters

$refLocalNeighbour
: Specifies a reference to the hash list that defines a local neighbor and for which list the remote neighbor is to be returned.

Description

The `GetRemoteNeighbours` method returns an array of remote neighbors associated with the specified local neighbor. The local neighbor is specified in the hash list passed to the *$refLocalNeighbour* parameter.

Example Usage

```
@remoteNeighbours = $NE->GetRemoteNeighbours($refLocalNeighbour);
```

Returns

Upon completion, the `GetRemoteNeighbours` method returns an array of remote neighbors (as a reference to a hash list).

Print

The `Print` method prints the current record.

Method Synopsis

```
Print()
```

Parameters

None

Description

The `Print` method prints the current record.

Example Usage

```
$NE->Print();
```

Returns

Upon completion, the `Print` method does not return any records.

RIV::RecordCache module reference

The `RIV::RecordCache` module provides an interface to access a record cache file.

The `RIV::RecordCache` module provides a constructor that creates and initializes a `RIV::RecordCache` file object. After creating this object, you can call methods that:

- Create or open an existing `RIV::RecordCache` file object
- Add a record to this `RIV::RecordCache` file object and obtain the key under which this record was added
- Retrieve all the records that reside in this `RIV::RecordCache` file object
- Retrieve a specific record from this `RIV::RecordCache` file object using the record's associated key

The constructor and methods are described in reference (man) page format.

RIV::RecordCache module synopsis

The `RIV::RecordCache` module synopsis shows how to make calls to the constructor and record cache operation methods that this module provides.

```
use RIV::RecordCache;
  $recordCache = new RIV::RecordCache($rivSession, $cacheName [ ,$cacheLocation ] );
  my $recKey = $recordCache->CacheRecord($record);
  $recordCache->GetRecords();
  $recordCache->GetRecord($recKey);
```

RIV::RecordCache Constructor

The `RIV::RecordCache` constructor creates and initializes a new `RIV::RecordCache` file object.

Constructor

```
new($rivSession, $cacheName [,$cacheLocation])
```

Parameters

$rivSession
: Specifies a blessed reference to either a `RIV::App` or `RIV::Agent` object. More specifically, this is a `RIV::App` or `RIV::Agent` application object returned in a previous call to the `RIV::App` or `RIV::Agent` constructor.

$cacheName
: Specifies the name of the `RIV::RecordCache` file object to be created or read from.

$cacheLocation
: Specifies the path to the `RIV::RecordCache` file object.

  This parameter is optional. If you do not pass a value to this parameter, the path to the `RIV::RecordCache` file object is assumed to be the `$NCHOME/var/precision` directory.

Description

The `RIV::RecordCache` constructor creates and initializes a new `RIV::RecordCache` file object with the name as specified in the *$cacheName* parameter and the location as specified in the *$cacheLocation* parameter.

Example Usage

The following code fragment illustrates a typical call to the `RIV::RecordCache` constructor:

```
$app = RIV::App::new();
 $cache = RIV::RecordCache::new($app, "Disco.Cache.Details.returns.MYDOMAIN",
                                "/opt/netcool/var/precision/");
}
```

Returns

Upon completion, the `RIV::RecordCache` constructor returns a `RIV::RecordCache` file object. This is the object upon which you can perform add and retrieve record operations.

See Also

- "RIV module reference" on page 53
- "RIV::Agent Constructor" on page 73
- "RIV::App Constructor" on page 95
- "CacheRecord" on page 120
- "GetRecord" on page 120
- "GetRecords" on page 121

CacheRecord

The `CacheRecord` method attempts to add the specified record to the specified cache.

Method Synopsis

`CacheRecord(`*`$record`*`)`

Parameters

$record
: Specifies the record that is to be added to the cache. This record is expressed as a hash.

Description

The `CacheRecord` method adds the record specified in the *$record* parameter to the specified cache. You specified the name of the cache in a previous call to the `RIV::RecordCache` constructor.

Example Usage

The following example illustrates a typical call to the `CacheRecord` method, where the method caches the record (hash) called *$myRec*:

```
$cache->CacheRecord($myRec);
```

Returns

Upon completion, the `CacheRecord` method returns:

- The value -1 to indicate that the attempt to add the record to the cache was unsuccessful. The method displays an appropriate error message requesting that you check to ensure that the cache is valid.
- The key that the record was added under if the attempt to add the record to the cache was successful.

See Also

- "RIV::RecordCache Constructor" on page 119

GetRecord

The `GetRecord` method retrieves from the cache a record associated with the specified key.

Method Synopsis

`GetRecord(`*`$recordKey`*`)`

Parameters

$recordKey
: Specifies the key associated with the record to be retrieved from the cache. This key was returned in a previous call to the `CacheRecord` method after it successfully inserted the record into the cache.

Description

The `GetRecord` method retrieves from the cache a record associated with the key specified in the *$recordKey* parameter. You specified the name of the cache in a previous call to the `RIV::RecordCache` constructor.

Example Usage

The following example illustrates a typical call to the `GetRecord` method, where the method returns to *$record* a hash from a previously specified cache that contains several records:

```
my $record = $cache->GetRecord();
```

Returns

Upon completion, the `GetRecord` method returns:

- *%record* — Specifies a hash that represents one of the records residing in the cache.

See Also

- "RIV::RecordCache Constructor" on page 119
- "CacheRecord" on page 120
- "GetRecords"

GetRecords

The `GetRecords` method retrieves from the cache a list of all the records currently residing in it.

Method Synopsis

```
GetRecords()
```

Parameters

None

Description

The `GetRecords` method retrieves from the cache a list of all the records currently residing in it. Each record is returned as a hash within a list.

Example Usage

The following example illustrates a typical call to the `GetRecords` method, where the method returns to *recordList* an array of hashes from a previously specified cache that contains several records:

```
my @recordList = $cache->GetRecords();
```

Returns

Upon completion, the `GetRecords` method returns:

- *$recordList*— Specifies an array of hashes, where each hash represents one of the records in the cache.

See Also

- "RIV::RecordCache Constructor" on page 119
- "CacheRecord" on page 120
- "GetRecord" on page 120

RIV::SnmpAccess module reference

The `RIV::SnmpAccess` module provides an interface to perform SNMP-related operations on Network Manager MIB trees.

The `RIV::SnmpAccess` module provides a constructor that allows you to create and initialize a new `RIV::SnmpAccess` session object. After obtaining this session object, you can call the synchronous or asynchronous versions of the `SnmpGet`-related methods to perform the following operations:

- SNMP `get`
- SNMP `get-next`
- SNMP `get-bulk`

The `RIV::SnmpAccess` module also provides several utility methods that allow you to operate on ANS.1 (Abstract Syntax Notation One) values and the MIB tree.

Note: Discovery agents implemented with this version of the Perl API should use the SNMP methods that the `RIV::Agent` module provides to obtain SNMP information from a network device.

The constructor and methods are described in reference (man) page format.

RIV::SnmpAccess module synopsis

The `RIV::SnmpAccess` module synopsis shows how to make calls to the constructor and SNMP operation methods that this module provides.

Synopsis

```
use RIV::SnmpAccess;
$RIV::SnmpAccess::MaxAsyncConcurrent;
$snmp = new RIV::SnmpAccess($RivSession);
(\%varop) = $snmp->SnmpGet($host, $addOn, $oid [, $instance,
$splitOutput]);
$ok = $snmp->AsyncSnmpGet($tag, $host, $addOn, $oid [, $instance]);
(\@varops) = $snmp->SnmpGetNext($host, $addOn, $oid, $instance]);
$ok = $snmp->AsyncSnmpGetNext($tag, $host, $addOn, $oid [, $instance]);
(\@varops) = $snmp->SnmpGetBulk($host, $addOn, \@oidList, $nonRepeat,
$maxRepeat, [, $instance, $splitOutput]);
$ok = $snmp->AsyncSnmpGetBulk($tag, $host, $addOn, \@oidList, $nonRepeat,
$maxRepeat, [, $instance, $splitOutput]);
where:

$asn1 = $varop{ASN1};
$value = $varop{VALUE};
foreach my $vp (@varops) {
$asn1 = $vp->{ASN1};
$value = $vp->{VALUE};
...
}
($baseOid, $indexOid, $baseOidName) = $snmp->SplitOidAndIndex($fullASN1);
$asn1 = $snmp->OidToASN1($mibIdentifier);
```

RIV::SnmpAccess Constructor

The `RIV::SnmpAccess` constructor creates and initializes a new `RIV::SnmpAccess` object.

Constructor

`new(`*`$rivSession`*`)`

Parameters

$rivSession
: Specifies a blessed reference to either a `RIV::App` or `RIV::Agent` object. More specifically, this is a `RIV::App` or `RIV::Agent` application object returned in a previous call to the `RIV::App` or `RIV::Agent` constructor.

Description

The `RIV::SnmpAccess` constructor creates and initializes a new `RIV::SnmpAccess` session object that must be a blessed reference to either a `RIV::App` or `RIV::Agent` object.

You can create only one `RIV::SnmpAccess` session object in any Perl application. If multiple domains are being supported (that is, multiple `RIV::App` objects) one of the application sessions must be used as the base for the `RIV::SnmpAccess` session.

Example Usage

```
$app = new RIV::App();
$snmp = new RIV::SnmpAccess('TEST', 'ncp_test');
```

Returns

Upon completion, the `RIV::SnmpAccess` constructor returns a `RIV::SnmpAccess` session object.

See Also

- "RIV::Agent Constructor" on page 73
- "RIV::App Constructor" on page 95

ASN1ToOid

The `ASN1ToOid` method converts the specified ASN.1 value to its corresponding OID.

Method Synopsis

`ASN1ToOid(`*`$asn1`*`)`

Parameters

$ans1 Specifies the ASN.1 (Abstract Syntax Notation One) value to be converted to its corresponding object identifier (OID).

Description

The `ASN1ToOid` method converts the specified ASN.1 value (*$ans1*) to its corresponding OID.

Example Usage

The following example should return *$oid* as `ifIndex`.

```
$oid = $snmp->ASN1ToOid(1.3.6.1.2.1.2.2.1.11.3.6.1.2.1.2.2.1.1)
```

Returns

Upon completion, the `ASN1ToOid` method returns an OID that corresponds to the specifiedASN.1 value (*$ans1*) value.

See Also

- "RIV::SnmpAccess Constructor" on page 123

AsyncSnmpGet

The `AsyncSnmpGet` method performs an asynchronous SNMP `get` operation on the specified MIB variable.

Method Synopsis

```
AsyncSnmpGet($tag, $nodeIP, $addOn,
$oid [,$instance, $splitOutput])
```

Parameters

$tag Specifies a string that the `AsyncSnmpGet` method appends to `SNMP_`*$tag*. This tag is associated with the results of an SNMP `get` operation. For example, if you specify the string `GET` to the *$tag* parameter, the `AsyncSnmpGet` method associates the tag `SNMP_GET` with the results for this SNMP `get` operation.

$nodeIP
Specifies a valid host IP address.

$addOn
Specifies the suffix to the community string.

$oid Specifies the MIB variable for which you want to perform an asynchronous SNMP `get` operation.

$instance
Specifies the start of the MIB subtree to retrieve. You must specify *$instance* as an ASN1 string (for example, `5.3.15`).

This parameter is optional.

$splitOutput
Specifies a value of true or false. If set to true (1) returns three extra keys — `OID`, `INDEX`, and `NAMEMIB`. The default is false (0), that is, does not return the three extra keys.

This parameter is optional.

Description

The `AsyncSnmpGet` method performs an asynchronous SNMP `get` operation on the specified MIB variable (the*$oid* parameter). The `AsyncSnmpGet` method returns the three extra keys — `OID`, `INDEX`, and `NAMEMIB` — only if the *$splitOutput* parameter is set to true (1).

Example Usage

```
$snmp->AsyncSnmpGet('GET', $nodeIP, "", "ifDescr", "2", 1);
($tag, $data) = $snmp->RIV::GetResult(-1);
```

Returns

Upon successful completion, the `AsyncSnmpGet` method returns the value */%varop* and the tag `SNMP_`*$tag*. If the request failed, `AsyncSnmpGet` returns `undef`. The return value, along with the tag `SNMP_`*$tag*, are returned in a call to `RIV::GetResult`.

See Also

- "RIV::GetResult" on page 63
- "RIV::SnmpAccess Constructor" on page 123
- "MaxAsyncConcurrent" on page 128

AsyncSnmpGetBulk

The `AsyncSnmpGetBulk` method performs an asynchronous SNMP `get-bulk` operation on all MIB objects in the specified MIB table.

Method Synopsis

```
AsyncSnmpGetBulk($tag, $nodeIP, $addOn,
$oidBindList, $nonRepeaters, $maxRepetitions
[,$instance, $splitOutput])
```

Parameters

$tag Specifies a string that the `AsyncSnmpGetBulk` method appends to `SNMP_`*$tag*. This tag is associated with the results of an SNMP `get-bulk` operation. For example, if you specify the string `GETBULK` to the *$tag* parameter, the `AsyncSnmpGetBulk` method associates the tag `SNMP_GETBULK` with the results for this SNMP `get-bulk` operation.

$nodeIP
Specifies a valid a host IP address.

$addOn
Specifies the suffix to the community string.

$oidBindList
Specifies a reference to an array that contains the MIB variables for which you want to perform an asynchronous SNMP `get-bulk` operation. The following is an example of an array that contains two MIB variables:

```
$oidBindList = \@oids;
where,
@oids = ('sysDescr', 'ifIndex');
```

$nonRepeaters
Specifies the number of MIB variables at the start of the list of `@oids` that return a single value. In the previous example, the `@oids` list contains two MIB variables: `sysDescr` and `ifIndex`. Only the `sysDescr` MIB variable returns a single value. Thus, this parameter would be set to the value 1 for the previous example.

$maxRepetitions
Specifies the number of MIB variable values in the table to be returned. For example, if you specify the value 2 to the *$maxRepetitions* parameter, the `AsyncSnmpGetBulk` method returns only the values for the first two MIB

variables in the table. To return values for all MIB variables in the table, specify a large number for this parameter.

This parameter is relevant for MIB variables that return a table, for example, `ifIndex`.

$instance
Specifies the start of the MIB subtree to retrieve. You must specify *$instance* as an ASN1 string (for example, `5.3.15`).

This parameter is optional.

$splitOutput
Specifies a value of true or false. If set to true (1) returns three extra keys — `OID`, `INDEX`, and `NAMEMIB`. The default is false (0), that is, does not return the three extra keys.

This parameter is optional.

Description

The `AsyncSnmpGetBulk` method performs an asynchronous SNMP `get-bulk` operation on all MIB objects specified in *$oidBindList*. The `SnmpGetBulk` method returns the three extra keys — `OID`, `INDEX`, and `NAMEMIB` — only if the *$splitOutput* parameter is set to true (1).

Notes

The parameters *$nonRepeaters* and *$maxRepetitions* must be defined. No default values are specified for these parameters.

Example Usage

```
@oids=('sysDescr', 'sysContact', 'sysUpTime', 'ipInReceives',
'ipOutRequests', 'ipOutDiscards', 'ipForwDatagrams', 'tcpCurrEstab',
'ifDescr';
$snmp->AsyncSnmpGetBulk("GETBULK", $nodeIP, "", \@oids, 8, 100);
($tag, $data) = $snmp->RIV::GetResult(-1);
```

Returns

Upon completion, the `AsyncSnmpGetBulk` method returns a reference to an array of *varops* and the tag `SNMP_`*$tag*. If the request failed, `AsyncSnmpGetBulk` returns `undef`. The return value, along with the tag `SNMP_`*$tag*, are returned in a call to `RIV::GetResult`.

See Also

- "RIV::GetResult" on page 63
- "RIV::SnmpAccess Constructor" on page 123
- "MaxAsyncConcurrent" on page 128

AsyncSnmpGetNext

The `AsyncSnmpGetNext` method performs an asynchronous SNMP `get-next` operation on the specified MIB variable.

Method Synopsis

```
AsyncSnmpGetNext($tag, $nodeIP, $addOn,
$oid [,$instance, $splitOutput])
```

Parameters

$tag Specifies a string that the `AsyncSnmpGetNext` method appends to `SNMP_`*$tag*. This tag is associated with the results of an SNMP `get-next` operation. For example, if you specify the string `GETNEXT` to the *$tag* parameter, the `AsyncSnmpGetNext` method associates the tag `SNMP_GETNEXT` with the results for this SNMP `get-next` operation.

$nodeIP
Specifies a valid host IP address.

$addOn
Specifies the suffix to the community string.

$oid Specifies the MIB variable for which you want to perform an asynchronous SNMP `get-next` operation.

$instance
Specifies the start of the MIB subtree to retrieve. You must specify *$instance* as an ASN1 string (for example, `5.3.15`).

This parameter is optional.

$splitOutput
Specifies a value of true or false. If set to true (1) returns three extra keys — `OID`, `INDEX`, and `NAMEMIB`. The default is false (0), that is, does not return the three extra keys.

This parameter is optional.

Description

The `AsyncSnmpGetNext` method performs an asynchronous SNMP `get-next` operation on the specified MIB variable (*$oid*). The `AsyncSnmpGetNext` method returns the three extra keys — `OID`, `INDEX`, and `NAMEMIB` — only if the *$splitOutput* parameter is set to true (1).

Example Usage

```
$snmp->AsyncSnmpGetNext('GETNEXT', $nodeIP, "", "ifDescr");
($tag, $data) = $snmp->RIV::GetResult(-1);
```

Returns

Upon successful completion, the `AsyncSnmpGetNext` method returns a reference to an array of *varops* and the tag `SNMP_`*$tag*. If the request failed, `AsyncSnmpGetNext` returns `undef`. The return value, along with the tag `SNMP_`*$tag*, are returned in a call to `RIV::GetResult`.

See Also

- "RIV::GetResult" on page 63
- "RIV::SnmpAccess Constructor" on page 123

- “MaxAsyncConcurrent”

GetMibHash

The `GetMibHash` method gets the entire MIB tree.

Method Synopsis

```
GetMibHash()
```

Parameters

None

Description

The `GetMibHash` method gets the entire MIB tree by browsing the files that exist in the `$NCHOME/mibs` directory.

Example Usage

```
%tree=$snmp->GetMibHash();
```

Returns

Upon completion, the `GetMibHash` method returns the complete MIB tree constructed as a result of browsing the files that reside in the `$NCHOME/mibs` directory.

See Also

- “RIV::SnmpAccess Constructor” on page 123

MaxAsyncConcurrent

The `MaxAsyncConcurrent` package variable sets the maximum number of concurrent asynchronous requests.

Variable Synopsis

```
$RIV::SnmpAccess::MaxAsyncConcurrent
```

Description

The `MaxAsyncConcurrent` package variable sets the maximum number of concurrent asynchronous requests. The default is ten concurrent asynchronous requests. The value of this variable is used when the first asynchronous request is executed. Thereafter, any changes to this package variable are ignored.

You use this package variable with the following asynchronous methods:

- `AsyncSnmpGet`
- `AsyncSnmpGetNext`
- `AsyncSnmpGetBulk`

See Also

- “AsyncSnmpGet” on page 124
- “AsyncSnmpGetNext” on page 127
- “AsyncSnmpGetBulk” on page 125

OidToASN1

The `OidToASN1` method converts the specified OID to its corresponding ASN.1 value.

Method Synopsis

`OidToASN1(`*`$oid`*`)`

Parameters

$oid Specifies the object identifier (OID) to be converted to its corresponding ASN.1 (Abstract Syntax Notation One) value.

Description

The `OidToASN1` method converts the specified OID (*$oid*) to its corresponding ASN.1 value.

Example Usage

```
$asn1 = $snmp->OidToASN1('ifDescr');
```

Returns

Upon completion, the `OidToASN1` method returns an ASN.1 value that corresponds to the specified OID (*$oid*).

See Also

- "RIV::SnmpAccess Constructor" on page 123

SnmpGet

The `SnmpGet` method performs an SNMP `get` operation on the specified MIB variable.

Method Synopsis

```
SnmpGet($nodeIP, $addOn, $oid
[,$instance, $splitOutput])
```

Parameters

$nodeIP
Specifies a valid host IP address.

$addOn
Specifies the suffix to the community string.

$oid Specifies the MIB variable for which you want to perform an SNMP `get` operation.

$instance
Specifies the start of the MIB subtree to retrieve. You must specify *$instance* as an ASN1 string (for example, `5.3.15`).

This parameter is optional.

$splitOutput
Specifies a value of true or false. If set to true (1) returns three extra keys — `OID`, `INDEX`, and `NAMEMIB`. The default is false (0), that is, does not return the three extra keys.

This parameter is optional.

Description

The `SnmpGet` method performs an SNMP `get` operation on the specified MIB variable (*$oid*). The `SnmpGet` method returns the three extra keys — `OID`, `INDEX`, and `NAMEMIB` — only if the *$splitOutput* parameter is set to true (1).

Example Usage

```
$vap = $snmp->SnmpGet($nodeIP, "", "ifDescr", 1);
print "$vap->{ASN1}, $vap->{VALUE}", "\n";
```

Returns

Upon completion, the `SnmpGet` method returns */%varop*, where the *%varop* keys are `ASN1` and `VALUE`.

See Also

- "RIV::SnmpAccess Constructor" on page 123

SnmpGetBulk

The `SnmpGetBulk` method performs an SNMP `get-bulk` operation on all MIB objects in the specified MIB table.

Method Synopsis

```
SnmpGetBulk($nodeIP, $addOn, $oidBindList,
$nonRepeaters, $maxRepetitions
[,$instance, $splitOutput])
```

Parameters

$nodeIP
Specifies a valid host IP address.

$addOn
Specifies the suffix to the community string.

$oidBindList
Specifies a reference to an array that contains the MIB variables for which you want to perform an SNMP `get-bulk` operation. The following is an example of an array that contains two MIB variables:

```
$oidBindList = \@oids;
where,
@oids = ('sysDescr', 'ifIndex');
```

$nonRepeaters
Specifies the number of MIB variables at the start of the list of `@oids` that return a single value. In the previous example, the `@oids` list contains two MIB variables: `sysDescr` and `ifIndex`. Only the `sysDescr` MIB variable returns a single value. Thus, this parameter would be set to the value 1 for the previous example.

$maxRepetitions
Specifies the number of MIB variable values in the table to be returned. For example, if you specify the value 2 to the *$maxRepetitions* parameter, the `SnmpGetBulk` method returns only the values for the first two MIB variables in the table. To return values for all MIB variables in the table, specify a large number for this parameter.

This parameter is relevant for MIB variables that return a table, for example, `ifIndex`.

$instance
: Specifies the start of the MIB subtree to retrieve. You must specify *$instance* as an ASN1 string (for example, `5.3.15`).

  This parameter is optional.

$splitOutput
: Specifies a value of true or false. If set to true (1) returns three extra keys — `OID`, `INDEX`, and `NAMEMIB`. The default is false (0), that is, does not return the three extra keys.

  This parameter is optional.

Description

The `SnmpGetBulk` method performs an SNMP `get-bulk` operation on all MIB objects specified in *$oidBindList*. The `SnmpGetBulk` method returns the three extra keys — `OID`, `INDEX`, and `NAMEMIB` — only if the *$splitOutput* parameter is set to true (1).

Notes

The parameters *$nonRepeaters* and *$maxRepetitions* must be defined. No default values are specified for these parameters.

Example Usage

```
@oids=('sysDescr','sysContact','sysUpTime','ipInReceives',
'ipOutRequests','ipOutDiscards','ipForwDatagrams','tcpCurrEstab',
'ifDescr');
($vap) = $snmp->SnmpGetBulk($nodeIP, "", \@oids, 8, 100);
```

Returns

Upon completion, the `SnmpGetBulk` method returns a reference to a result array. Each element of the result array is a `%varop` hash.

See Also

- "RIV::SnmpAccess Constructor" on page 123

SnmpGetNext

The `SnmpGetNext` method performs an SNMP `get-next` operation on the specified MIB variable.

Method Synopsis

```
SnmpGetNext($nodeIP, $addOn, $oid,
[$instance, $splitOutput])
```

Parameters

$nodeIP
: Specifies a valid IP address.

$addOn
: Specifies the suffix to the community string.

$oid Specifies the MIB variable for which you want to perform an SNMP `get-next` operation.

$instance

Specifies the start of the MIB subtree to retrieve. You must specify *$instance* as an ASN1 string (for example, `5.3.15`).

This parameter is optional.

$splitOutput

Specifies a value of true or false. If set to true (1) returns three extra keys — `OID`, `INDEX`, and `NAMEMIB`. The default is false (0), that is, does not return the three extra keys.

This parameter is optional.

Description

The `SnmpGetNext` method performs iterative SNMP `get-next` operations on the specified host (*$nodeIP*) for the MIB table starting at the specified MIB variable (*$oid*). The `SnmpGetNext` method returns the three extra keys — `OID`, `INDEX`, and `NAMEMIB` — only if the *$splitOutput* parameter is set to true (1).

Example Usage

```
($vap) = $snmp->SnmpGetNext($nodeIP, "", "ifDescr");
```

Returns

Upon completion, the `SnmpGetNext` method returns a reference to a result array. Each element of the result array is a *%varop* hash.

See Also

- "RIV::SnmpAccess Constructor" on page 123

SplitOidAndIndex

The `SplitOidAndIndex` method converts the full ASN.1 value into its index and the base OID.

Method Synopsis

```
SplitOidAndIndex($fullASN1)
```

Parameters

$fullASN1

Specifies the complete ASN.1 (Abstract Syntax Notation One) value to be split.

Description

The `SplitOidAndIndex` method splits the specified ASN.1 value (*$fullASN1*) into its index and the base OID (object identifier).

Example Usage

The following call to `SplitOidAndIndex` passes an ASN.1 value of `1.3.6.1.2.1.2.2.1.2.0` to the *$fullASN1* parameter:

```
($baseOid, $indexOid, $baseOidName) = $snmp->SplitOidAndIndex($fullASN1);
```

The previous call returns the following values:

- *$baseOID*=`1.3.6.1.2.1.2.2.1.2`
- *$indexOID*=`0`
- *$baseOidName*=`ifDescr`

Returns

Upon completion, the `SplitOidAndIndex` method returns an array with three elements:

- The base OID.
- The index.
- The name of the base OID.

See Also

- "RIV::SnmpAccess Constructor" on page 123

Appendix B. NCP Modules Reference

Each NCP module provides constructors and methods used in the Perl scripts that you implement to perform operations on NCIM topology databases and NCIM domains.

To implement Perl scripts using the NCP modules, you must be familiar with the constructors and methods that each module provides. These constructors and methods are described in manual (reference) page format.

The following list identifies the NCP modules:

- `NCP::DBI_FACTORY`
- `NCP::Domain`

NCP::DBI_Factory module reference

The `NCP::DBI_Factory` module provides an interface to make it easier to use the standard Perl DBI module to perform operations on NCIM topology databases.

The `NCP::DBI_Factory` module provides a method used to create a standard DBI handle used in subsequent calls to some of the methods that perform operations on NCIM topology databases.

Use of the methods that the `NCP::DBI_Factory` module provides assumes that you understand how to use the standard Perl DBI module and that you are familiar with NCIM topology databases.

See *IBM Tivoli Network Manager IP Edition Topology Database Reference* (SC27-2766-00) for information on NCIM topology databases.

Each of the `NCP::DBI_Factory` module methods is described in manual (reference) page format.

NCP::DBI_Factory module synopsis

The `NCP::DBI_Factory` module synopsis shows how to make calls to some of the NCIM database operation methods that this module provides.

The comments provided in the synopsis serve as a quick reference as to the purpose of the NCIM database operation methods. The reference (man) pages provide the details.

```
# Load the NCP::DBI_Factory module.
use NCP::DBI_Factory;

# Get the database login details from DbLogins.NCOMS.cfg, or,
# failing that, from DbLogins.cfg.
     my %typicalParameters = (
                         domain => "NCOMS",
                         dbid => "NCIM",
                         );

# Call the createDbHandle method to obtain the DBI handle. In this call,
# pass the %typicalParameters hash parameter.
my $dbh = NCP::DBI_Factory::createDbHandle(%typicalParameters);
my %explicitParams = (
               dbname => "ncim",
               server => "mysql",
               schema => "ncim",
               host => "192.168.1.1",
               username => "dbuser",
               password => "dbpassword",
               port => 3406 # optional
             );

# Call the createDbHandle method to obtain a second DBI handle. In this call,
# pass the %explicitParameters hash parameter.
my $otherDbh = NCP::DBI_Factory::createDbHandle(%explicitParams);

# Declare variables that the insert_row and insert_auto_inc_row methods use.
my $tableName = "entityNameCache";
my $name = "entity1";

# Declare a hash that the insert_row and insert_auto_inc_row methods use.
# Note: The string in $name will automatically be quoted.
my %row = (
    entityName => $name,
    domainMgrId => 1
);

# Call the insert_row method to insert a row into a database table
# called entityNameCache.
NCP::DBI_Factory::insert_row($dbh, $tableName, \%row)
    or print "Insert failed ", $dbh->errstr "\n";

    my $autoIncColumnName = "entityId";

# Call the insert_auto_inc_row method to insert a row into a database table
# called entityNameCache. This table has an auto incremented column called
# entityId.
my $newId = NCP::DBI_Factory::insert_auto_inc_row(
    $dbh, $tableName, \%row, $autoIncColumnName)
    or print "Insert failed ", $dbh->errstr "\n";

# Set up the variabloes to use in the calls to the
# prepare_insert_auto_inc and execute_insert_auto_inc methods.
my @columnName = ["entityName","domainMgrId"];
my @values = ["entity2",2];
my $sth = NCP::DBI_Factory::prepare_insert_auto_inc(
    $dbh, $tableName, $autoIncColumnName, @columnNames)
    or print "Prepared failed ", $dbh->errstr, "\n";

my $newId2 = NCP::DBI_Factory::execute_insert_auto_inc(
    $dbh, $sth, @values)
    or print "Insert failed ", $dbh->errstr "\n";

$sth->finish();

# Commit the changes to the NCIM topology database by calling
# the Perl DBI module commit method. Otherwise, call the Perl DBI
# rollback method to undo the most recent series of uncommitted
# database changes.
if ($happy)
{
    $dbh->commit();
}
else
{
    $dbh->rollback();
}

# Identify the current schema by calling the schema method.
# Call the tables method to return a sorted array of table
# and view names for the current schema.
# Call the describeTable method to return a sorted array
# of upper case field names for the specified table
# in the current schema.
my $schema = NCP::DBI_Factory::schema(%typicalParameters);
my @tableList = NCP::DBI_Factory::tables(dbh => $dbh,
                                         schema => $schema,
                                         %typicalParameters);
foreach my $table (@tableList)
{
my @fields = NCP::DBI_Factory::describeTable(
                                         $table,
                                         dbh => $dbh,
                                         schema => $schema);
```

createDbHandle

The `createDbHandle` method creates a standard DBI handle, connected to the requested NCIM topology database. This DBI handle is used in subsequent calls to some of the other `NCP::DBI_Factory` module methods.

Method Synopsis

`NCP::DBI_Factory::createDbHandle(`*`%typicalParameters`*`)`

`NCP::DBI_Factory::createDbHandle(`*`%explicitParameters`*`)`

Parameters

%typicalParameters

Specifies a hash that contains the key/value pairs necessary for `createDbHandle` to access information from one of the following files in order to create a DBI handle:

- `DbLogins.cfg` — Specifies the standard database log-ins configuration file.
- `DbLogins.`*domain*`.cfg` — Specifies a domain-specific database log-ins configuration file where *domain* identifies a domain (for example, `DbLogins.NCOMS.cfg`).
- Custom file — Specifies an optional custom database log-ins configuration file. This file is expected to have the same format as `DbLogins.cfg` and `DbLogins.`*domain*`.cfg`.

The following table identifies the key/value pairs in this hash:

Hash key	Description
domain	Specifies the name of the domain used to identify whether a DbLogins.*domain*.cfg file exists in the $NCHOME/etc/precision directory. The following example shows a possible value for this key: `domain => "NCOMS"` In this example, the createDbHandle method would look for a file called DbLogins.NCOMS.cfg in the $NCHOME/etc/precision directory. If a DbLogins.*domain*.cfg file does not exist, createDbHandle looks for the DbLogins.cfg file. This is a required key/value pair.
dbid	Specifies the logical name for the NCIM topology database to which you want to connect. Each NCIM topology database has a unique logical name specified in the DbLogins.cfg, DbLogins.*domain*.cfg, or custom database log-ins configuration file. The following example shows a possible value for this key: `dbid => "ncim"` In this example, the value ncim specifies the logical name for this connection to the NCIM topology database. The createDbHandle method uses dbid to locate the appropriate section of the database log-ins configuration file. This is a required key/value pair. **Note:** The dbid key/value pair maps to the m_DbId field in the database log-ins configuration file.
dbfile	Specifies the name of the custom database log-ins configuration file. If you specify the optional dbfile key/value pair, the createDbHandle method would look for the specified custom file in the $NCHOME/etc/precision directory.

%explicitParameters

Specifies a hash that contains the key/value pairs necessary to create a DBI handle. In this case, createDbHandle does not obtain the necessary values from a file as is the case for the *%typicalParameters* hash parameter. Instead, all of the necessary values are explicitly specified. (Typically, an application would obtain these values from the command line.) The following table identifies the key/value pairs in this hash. All key/value pairs listed in the table are required, except for `port`, which is optional.

Hash key	Description
`dbname`	Specifies the name of the NCIM topology database to which you want to connect. The following example shows a possible value for this key: `dbname => "ncim"` In this example, the value `ncim` specifies that you want to connect to the NCIM topology database.
`server`	Specifies a string that identifies the type of database associated with the database name specified in the `dbname` key. The following list identifies the possible values for this key: • `mysql` — Specifies the MySQL database. • `oracle` — Specifies the Oracle database. • `db2` — Specifies the DB2 database. • `informix` — Specifies the Informix® database. The following example shows a database type of MySQL: `server => "mysql"`
`schema`	Specifies the name of the schema to access in the database specified in the `dbname` key. The following example shows a possible value for this key: `schema => "ncim"`
`host`	Specifies the address of the host computer on which the specified NCIM topology database resides. The following example shows a possible value for this key: `host => "192.168.1.1"`
`username`	Specifies the name of the user who has access to the specified NCIM topology database. The following example shows a possible value for this key: `username => "dbuser"`

Hash key	Description
`password`	Specifies the password of the user who has access to the specified NCIM topology database. The following example shows a possible value for this key: `password => "dbpassword"`
`port`	Specifies an optional key that identifies the port associated with the address specified in `host`. The following example shows a possible value for this key: `port => "3406"`

Description

The `createDbHandle` method creates a standard DBI (Database Interface) handle to be used in subsequent calls to some of the other `NCP::DBI_Factory` methods. This DBI handle contains the information needed to connect to the requested NCIM topology database.

The `createDbHandle` method accepts the following hash parameters:

- *%typicalParameters* — This hash provides the `domain` and `dbid` key/value pairs. Optionally, this hash can provide a `dbfile` key/value pair. Given this information, the `createDbHandle` method:
  – Reads and parses one of these files that resides in the `$NCHOME/etc/precision` directory: `DbLogins.cfg` (the default), `DbLogins.`*`domain`*`.cfg`, or an optional custom database login-ins configuration file.
  – Uses the `dbid` key/value pair to locate the database entry of interest in the specified database log-ins configuration file.
  – Connects to the specified NCIM topology database.
  – Sets the context to the schema associated with the specified NCIM topology database.
- *%explicitParameters* — This hash provides all of the required information from the command line. Given this information, the `createDbHandle` method:
  – Connects to the specified NCIM topology database.
  – Sets the context to the schema associated with the specified NCIM topology database.

When reading from a database log-ins configuration file, `createDbHandle` can override any values from the file if you explicitly pass them in from the command line. The following table provides the available override options and their mappings to the fields in the database log-ins configuration file:

Override option	Description
`dbfile`	Specifies an optional override for the `DbLogins.cfg`database log-ins configuration file. This file is expected to have the same format as `DbLogins.cfg` and `DbLogins.`*domain*`.cfg`.

Override option	Description
dbname	Specifies the name of the database. If specified on the command line, this option overrides the value specified for the m_DbName field in the database log-ins configuration file.
server	Specifies a string that identifies the type of database associated with the database name specified in the dbname option. The following list identifies the possible values for the server option: • mysql — Specifies the MySQL database. • oracle — Specifies the Oracle database. • db2 — Specifies the DB2 database. • informix — Specifies the Informix database. This option overrides the value specified for the m_Server field in the database log-ins configuration file.
schema	Specifies the name of the schema to access in the specified database. This option overrides the value specified for the m_Schema field in the database log-ins configuration file.
host	Specifies the address of the host computer on which the specified NCIM topology database resides. This option overrides the value specified for the m_Hostname field in the database log-ins configuration file.
username	Specifies the name of the user who has access to the specified NCIM topology database. This option overrides the value specified for the m_Username field in the database log-ins configuration file.
password	Specifies the password of the user who has access to the specified NCIM topology database. This option overrides the value specified for the m_Password field in the database log-ins configuration file.
port	Specifies the port associated with the address specified in host. This option overrides the value specified for the m_PortNum field in the database log-ins configuration file.

Notes

To ensure that the createDbHandle method can print appropriate messages to a log file, you must have previously specified a log handle (that is, a reference to a file object) by calling the setLogHandle method. Otherwise, the method sends these messages to STDOUT.

Example Usage

The following code example illustrates a typical call to the `createDbHandle` method using the *%typicalParameters* hash parameter:

```
# Set up the hash list to contain the domain and
# database ID.

my %typicalParameters = (
                            domain => "NCOMS",
                            dbid => "NCIM"
                            );

# Call the createDbHandle method passing to it the previously
# set up hash list. In this case, createDbHandle knows that the
# information it needs to create the DBI handle resides in a file.
# The createDbHandle method returns the DBI to the $dbh variable.
#

my $dbh = NCP::DBI_Factory::createDbHandle(%typicalParameters);
```

The following code example illustrates a typical call to the `createDbHandle` method using the *explicitParams* parameter:

```
# Set up the hash list to contain the information necessary to create
# the DBI handle without reading a database log-ins configuration file.

my %explicitParams = (
                   dbname => "ncim",
                   server => "mysql",
                   schema => "ncim",
                   host => "9.180.209.24",
                   username => "batman",
                   password => "robin",
                   port => 3406 # This is an optional element.
                 );

# Call the createDbHandle method passing to it the previously set up
# hash list that contains the information necessary to create the
# DBI handle. The createDbHandle method returns the DBI handle to
# the $otherDbh variable.
my $otherDbh = NCP::DBI_Factory::createDbHandle(%explicitParams);
```

Returns

Upon completion, the `createDbHandle` method returns a standard DBI handle associated with the requested NCIM topology database.

See Also

- “schema” on page 153

describeTable

The `describeTable` method returns a sorted array of uppercase field names for the specified table or view.

Method Synopsis

NCP::DBI_Factory::describeTable(*$tableName*, *%dbhschema*)

NCP::DBI_Factory::describeTable(*$tableName*, *%typicalParameters*)

Parameters

$tableName

Specifies the name of the database table that is of interest. Because the different databases that the `DBI_Factory` module supports use different cases for table names, supply the table name in mixed case. For example, if the table name is `entitynamecache`, then the mixed case equivalent is `entityNameCache`.

In either case, the `describeTable` method internally converts the specified database table name to upper or lower case as required.

%dbhschema

Specifies a hash that contains the following keys:

- `dbh` — Specifies an existing DBI handle returned in a previous call to the `createDbHandle` method. This handle supplies the context for connecting to the specified NCIM topology database.
- `schema` — Specifies the schema that contains the database table name specified in the *$tableName* parameter. Typically, this schema name is obtained in a call to the `schema` method.

%typicalParameters

Specifies the same hash parameter accepted by the `createDbHandle` method.

Description

The `describeTable` method returns a sorted array of uppercase field names for the database table or view specified in the *$tableName* parameter. For full portability, pass this table name in mixed case to the *tableName* parameter. The `describeTable` method:

- Converts the table name to upper case for Oracle and DB2 databases, since these databases require upper case table names. For example, the table name `entityNameCache` would be converted to `ENTITYNAMECACHE`.
- Accepts the mixed case table name for a MySql database, since this database requires mixed case table names. For example, the table name `entityNameCache` would be accepted as `entityNameCache`.
- Accepts the mixed case table name for an Informix database. This type of database accepts either upper or mixed case.

If you specify the *%typicalParameters* hash instead of the *%dbhschema* hash, `describeTable` calls `createDbHandle` to create a new DBI handle. The schema associated with this newly created handle is identified by the `dbid` key/value pair and this schema is expected to contain the table specified in the *$tableName* parameter.

Notes

The `NCP::DBI_Factory` module supports DB2, Oracle, Informix, and MySql databases. In all of these databases, tables and field names are case insensitive with regard to SQL statements. However, note the following about field names in table rows returned by these databases:

- DB2 and Oracle — Return field names in uppercase.
- Informix — Returns field names in lowercase.
- MySql — Returns field names in mixed case.

Example Usage

The following code example illustrates a typical call to the describeTable method using the *%dbhschema* hash parameter. The code example also shows a call to the tables method:

```
# List the tables in schema $schema without creating a new DBI handle.
my @tableList = NCP::DBI_Factory::tables(dbh => $dbh,
                                         schema => $schema);
 foreach my $table (@tableList)
  {
     my @fields = NCP::DBI_Factory::describeTable(
                                          $table,
                                          dbh => $dbh,
                                          schema => $schema);
    }
```

Returns

Upon completion, the describeTable method returns a sorted array of uppercase field names for the specified database table or view.

See Also

- "createDbHandle" on page 137
- "schema" on page 153
- "tables" on page 156

execute_insert_auto_inc

The execute_insert_auto_inc method executes an auto-incremented column statement handle prepared by the prepare_insert_auto_inc method.

Method Synopsis

```
NCP::DBI_Factory::execute_insert_auto_inc($dbHandle,$statementHandle,
 $values)
```

Parameters

$dbHandle
Specifies the DBI handle returned in a previous call to the createDbHandle method. This handle supplies the context for connecting to the specified NCIM topology database.

$statementHandel
Specifies the statement handle returned in a previous call to the prepare_insert_auto_inc method.

$values
Specifies the values to be executed.

Description

The execute_insert_auto_inc method executes an auto-incremented column statement handle prepared by the prepare_insert_auto_inc method.

Notes

To ensure that the execute_insert_auto_inc method can print appropriate messages to a log file, you must have previously specified a log handle (that is, a reference to a file object) by calling the `setLogHandle` method. Otherwise, the method sends these messages to `STDOUT`.

Example Usage

The following code example illustrates a typical call to the `execute_insert_auto_inc` method:

```
ToBeSupplied
```

Returns

Upon completion, the `execute_insert_auto_inc` method returns the new auto-incremented value.

See Also

- "createDbHandle" on page 137
- "prepare_insert_auto_inc" on page 152

extractCmdLineOptions

The `extractCmdLineOptions` method allows database login options specified on the command line to be provided in a common format.

Method Synopsis

`NCP::DBI_Factory::extractCmdLineOptions([`*`$prefix`*`])`

Parameters

$prefix
An optional parameter that specifies a prefix used to allow other similar database login options to be supplied for multiple database connections. Examples of such prefixes include `ncim_`, `ncmonitor_`, and `ncpoller_`.

Description

The `extractCmdLineOptions` method allows database login options specified on the command line to be provided in a common format. This method accepts the same database login options as the `createDbHandle` method:

- `dbfile`
- `server`
- `dbname`
- `schema`
- `host`
- `username`
- `password`
- `port`

The `extractCmdLineOptions` method can also take an optional *$prefix* parameter that specifies similar database login options other than the previously listed options. This optional parameter allows Perl scripts to handle multiple sets of database login options by calling the `extractCmdLineOptions` method multiple times.

Notes

The `extractCmdLineOptions` method removes the database login options that it processes and returns in the hash from the `@ARGV` array. However, any options that do not get processed and returned in the hash remain in the `@ARGV` array.

Use the `extractCmdLineOptions` method to process database login options from the command line. Use the `extractHashRefOptions` method to process database login options from a hash reference.

Example Usage

You can call the `extractCmdLineOptions` method with or without the *$prefix* parameter.

Calling extractCmdLineOption without the $prefix parameter

The following code example illustrates a call to the `extractCmdLineOptions` method without the use of the *$prefix* optional parameter. The example declares a variable called *$optionsHashRef* to store the reference to the hash returned by `extractCmdLineOptions`:

```
my $optionsHashRef = NCP::DBI_Factory::extractCmdLineOptions();
```

Assume that the previous code example is contained in a Perl script called `dboptions.pl`. Consider this script executed with the `password` , `host`, and `whatever` database login options:

```
dboptions.pl -password tom -host dick -whatever harry
```

The `extractCmdLineOptions` returns a reference to a hash in *$optionsHashRef* as follows:

```
$optionsHashRef = { password => "tom", host => "dick" }
```

The database login option — `( '-whatever', 'harry' )` — remains in the `@ARGV` array because the `extractCmdLineOptions` method could not process it without the *$prefix* optional parameter.

Calling extractCmdLineOption with the $prefix parameter

The following code example illustrates multiple calls to the `extractCmdLineOptions` method with the use of the *$prefix* optional parameter:

```
my $generic =
NCP::DBI_Factory::extractCmdLineOptions();
my $ncimSpecific =
NCP::DBI_Factory::extractCmdLineOptions("ncim_")|| $generic;
my $ncmonitorSpecific =
NCP::DBI_Factory::extractCmdLineOptions("ncmonitor_") || $generic;
my $ncpollerSpecific =
NCP::DBI_Factory::extractCmdLineOptions("ncpoller_") || $generic;
```

Assume that the previous code example is contained in a Perl script called `dboptionsuseprefix.pl`. Consider this script executed with the `password` and `ncpoller_password` database login options:

```
dboptionsuseprefix.pl -password "ncim" -ncpoller_password "ncpoller"
```

The `extractCmdLineOptions` returns references to hashes in *$ncimSpecific*, *$ncmonitorSpecific*, and *$ncpollerSpecific* as follows:

```
$ncimSpecific = { password => "ncim" };
$ncmonitorSpecific = { password => "ncim" };
$ncpollerSpecific = { password => "ncpoller" };
```

The following list further explains how these calls to `extractCmdLineOptions` work:

- The first call to `extractCmdLineOptions` (without the optional *$prefix* parameter) processes the `-password "ncim` database login option and returns a reference to a hash that contains `{ password => "ncim" }`.
- The second call to `extractCmdLineOptions` sets up a logical OR operation. If a database login option beginning with the prefix `ncim_` is specified, then process it and return the appropriate value in the hash reference. Otherwise, return `{ password => "ncim" }` to the hash reference. In this case, the right side of the logical OR is true.
- The third call to `extractCmdLineOptions` sets up a logical OR operation. If a database login option beginning with the prefix `ncmonitor_` is specified, then process it and return the appropriate value in the hash reference. Otherwise return `{ password => "ncim" }` to the hash reference. In this case, the right side of the logical OR is true.
- The fourth call to `extractCmdLineOptions` sets up a logical OR operation. If a database login option beginning with the prefix `ncpoller_` is specified, then process it and return the appropriate value in the hash reference. Otherwise return `{ password => "ncim" }` to the hash reference. In this case, the left side of the logical OR is true and so `password => "ncpoller` is returned.

Returns

Upon completion, the `extractCmdLineOptions` method returns a reference to a hash that contains the extracted database login options and values in key/value format. If no database login options were specified, the `extractCmdLineOptions` method returns `undef`.

See Also

- "createDbHandle" on page 137
- "extractHashRefOptions"

extractHashRefOptions

The `extractHashRefOptions` method extracts the database login options from the specified hash reference.

Method Synopsis

`NCP::DBI_Factory::extractHashRefOptions(`*`$originalHashRef`*` [,`*`$prefix`*`])`

Parameters

$originalHashRef
: Specifies a reference to the original hash that contains the database login options.

$prefix
: An optional parameter that specifies a prefix used to allow other similar database login options to be supplied for multiple database connections. Examples of such prefixes include `ncim_`, `ncmonitor_`, and `ncpoller_`.

Description

The extractHashRefOptions method extracts the database login options from the hash reference specified in the *$originalHashRef* parameter. This method accepts the same database login options as the createDbHandle method:

- dbfile
- server
- dbname
- schema
- host
- username
- password
- port

The extractHashRefOptions method can also take an optional *$prefix* parameter that specifies similar database login options other than the previously listed options. This optional parameter allows Perl scripts to handle multiple sets of database login options by calling the extractHashRefOptions method multiple times.

Notes

The extractHashRefOptions method does not remove the key/value pairs from the hash reference specified in the *$originalHashRef* parameter.

Use the extractHashRefOptions method to process database login options from a hash reference. Use the extractCmdLineOptions method to process database login options from the command line or DbLogins.cfg file.

Example Usage

The following code example sets up a hash reference and then makes two calls to the extractHashRefOptions method:

```
my %original =
{ password => "topsecret", ncpoller_password => "classified, foo => "bar" };

my $generic = NCP::DBI_Factory::extractHashRefOptions(\%original);
my $ncpoller = NCP::DBI_Factory::extractHashRefOptions(\%original, "ncpoller_");
```

The following further explains how these calls to extractHashRefOptions work:

- The first call to extractHashRefOptions extracts the -password => "topsecret" database login option and returns it to *$generic*. This call to extractHashRefOptions cannot extract the other two options because this call did not specify ncpoller_ or foo_ in the optional *$prefix* parameter. The -password => "topsecret" option remains in the %original hash reference.
- The second call to extractHashRefOptions extracts the -password => "classified" database login option and returns it to *$ncpoller*. This call to extractHashRefOptions extracts -password => "classified" because of the ncpoller_ prefix passed to the *$prefix* parameter. The -password => "classified" option remains in the %original hash reference.
- The foo => "bar" option is not extracted and remains in the %original hash reference.

Returns

Upon completion, the `extractHashRefOptions` method returns a reference to a hash that contains the extracted database login options and values in key/value format. If no database login options were specified, the `extractHashRefOptions` method returns `undef`.

See Also

- "createDbHandle" on page 137
- "extractCmdLineOptions" on page 145

insert_auto_inc_row

The `insert_auto_inc_row` method inserts a row into the specified table that has an auto-increment column.

Method Synopsis

```
NCP::DBI_Factory::insert_auto_inc_row($dbHandle,$tableName,
$tableRow, $autoIncColumnName)
```

Parameters

$dbHandle
Specifies the DBI handle returned in a previous call to the `createDbHandle` method. This handle supplies the context for connecting to the specified NCIM topology database.

$tableName
Specifies the name of the table into which the `insert_auto_inc_row` method inserts the row specified in the *$tableRow* parameter.

$tableRow
Specifies a hash of scalars keyed on the column name.

$autoIncColumnName
Specifies the name of the auto-increment column in the specified table.

Description

The `insert_auto_inc_row` method inserts the row specified in the *$tableRow* parameter into the table specified in the *$tableName* parameter. The table is expected to contain the auto-increment column name specified in the `$autoIncColumnName` parameter.

Note: You can also call the DBI `rollback` interface to undo the most recent insert row change.

Notes

To ensure that the `insert_auto_inc_row` method can print appropriate messages to a log file, you must have previously specified a log handle (that is, a reference to a file object) by calling the `setLogHandle` method. Otherwise, the method sends these messages to `STDOUT`.

Use the `insert_auto_inc_row` method to insert rows in tables that have an auto-increment column. Use the `insert_row` method to insert rows in tables that have do not have an auto- increment column.

Example Usage

The following code example illustrates a typical call to the `insert_auto_inc_row` method:

```
my $tableName = "entityNameCache";
    my $name = "fred";

    # Note: the string in $name will automatically be quoted
    my %row = (
        entityName => $name,
        domainMgrId => 1
    );

my $autoIncColumnName = "entityId";

    my $newId = NCP::DBI_Factory::insert_auto_inc_row(
        $dbh, $tableName, \%row, $autoIncColumnName)
        or print "Insert failed ", $dbh->errstr "\n";

    # Changes only take effect when this is called
    if ($happy)
    {
        $dbh->commit();
    }
    else
    {
        $dbh->rollback();
    }
```

Returns

Upon completion, the `insert_auto_inc_row` method returns the new auto-increment value, provided that the row could be uniquely identified by the fields that the `insert_auto_inc_row` method just inserted into the specified table.

See Also

- "createDbHandle" on page 137
- "insert_row"

insert_row

The `insert_row` method inserts a row into the specified table.

Method Synopsis

`NCP::DBI_Factory::insert_row(`*`$dbHandle`*`,`*`$tableName`*`, `*`$tableRow`*`)`

Parameters

$dbHandle
Specifies the DBI handle returned in a previous call to the `createDbHandle` method. This handle supplies the context for connecting to the specified NCIM topology database.

$tableName
Specifies the name of the table into which the `insert_row` method inserts the row specified in the *$tableRow* parameter.

$tableRow
Specifies a hash of scalars keyed on the column name.

Description

The `insert_row` method inserts the row specified in the *$tableRow* parameter into the table specified in the *$tableName* parameter. The `insert_row` method automatically interpolates any strings in *$tableRow* into double-quoted strings.

Note: You can also call the DBI `rollback` interface to undo the most recent insert row change.

Notes

To ensure that the `insert_row` method can print appropriate messages to a log file, you must have previously specified a log handle (that is, a reference to a file object) by calling the `setLogHandle` method. Otherwise, the method sends these messages to `STDOUT`.

Use the `insert_row` method to insert rows in tables that have do not have an auto-increment column. Use the `insert_auto_inc_row` method to insert rows in tables that have an auto- increment column.

Example Usage

The following code example illustrates a typical call to the `insert_row` method:

```
my $tableName = "entityNameCache";
    my $name = "fred";

    # Note: the string in $name will automatically be quoted
    my %row = (
        entityName => $name,
        domainMgrId => 1
    );

    NCP::DBI_Factory::insert_row($dbh, $tableName, \%row)
        or print "Insert failed ", $dbh->errstr "\n";

# Changes only take effect when commit is called
    if ($happy)
    {
        $dbh->commit();
    }
    else
    {
        $dbh->rollback();
    }
```

Returns

Upon completion, the `insert_row` method returns whatever the standard DBI statement handle execute method returns.

See Also

- "createDbHandle" on page 137
- "insert_auto_inc_row" on page 149

prepare_insert_auto_inc

The `prepare_insert_auto_inc` method prepares the SQL statement once so that it can be used multiple times when inserting many rows into an auto-increment column of the specified database table.

Method Synopsis

`NCP::DBI_Factory::prepare_insert_auto_inc(`*`$dbHandle`*`,`*`$tableName`*`,` *`$autoIncColumnName`*`,` *`$columnNames`*`)`

Parameters

$dbHandle
: Specifies the DBI handle returned in a previous call to the `createDbHandle` method. This handle supplies the context for connecting to the specified NCIM topology database.

$tableName
: Specifies the name of the table into which the `prepare_insert_auto_inc` method prepares the SQL statement to be inserted into multiple rows in the specified columns.

$autoIncColumnName
: Specifies the name of the auto-increment column in the specified table.

$columnNames
: Specifies a hash of column names.

Description

The `prepare_insert_auto_inc` method prepares the SQL statement once so that it can be used multiple times when inserting many rows into an auto-increment column of the specified database table. Use this method when inserting many rows into an auto-incremented column.

The returned SQL statement handle should be used with the `execute_insert_auto_inc` method.

Notes

To ensure that the `prepare_insert_auto_inc` method can print appropriate messages to a log file, you must have previously specified a log handle (that is, a reference to a file object) by calling the `setLogHandle` method. Otherwise, the method sends these messages to `STDOUT`.

Example Usage

The following code example illustrates a typical call to the `prepare_insert_auto_inc` method:

```
ToBeSupplied
```

Returns

Upon completion, the `prepare_insert_auto_inc` method returns the prepared SQL statement handle.

See Also

- "createDbHandle" on page 137
- "insert_row" on page 150

schema

The `schema` method returns the schema name associated with the specified database.

Method Synopsis

`NCP::DBI_Factory::schema(%`*`typicalParameters`*`)`

`NCP::DBI_Factory::schema(%`*`explicitParameters`*`)`

Parameters

$typicalParameters
: Specifies the same hash parameter accepted by the `createDbHandle` method. If you supply this hash parameter, `schema` obtains the value from the database log-ins configuration file.

%explicitParameters
: Specifies the same hash parameter accepted by the `createDbHandle` method. If you supply this hash parameter, `schema` obtains the value from the command line.

Description

The `schema` method returns the name of the schema being used as follows:

- If the *%typicalParameters* hash was specified — The `schema` method obtains the name of the schema being used from one of these files: `DbLogins.cfg`, `DbLogins.`*`DOMAIN`*`.cfg`, or an optional custom database log-ins configuration file. If `schema` finds a domain-specific file, it uses that file to obtain the name of the schema. If the *schema* variable was passed to the createDbHandle method, then the `schema` method uses this variable to obtain the name of the schema. The *schema* variable overrides the schema information contained in any of the configuration files.
- If the *%explicitParameters* hash was specified — The `schema` method obtains the name of the schema from the command line. (The schema name is the value associated with the `schema` key in the *%explicitParameters* hash.)

Notes

To ensure that the `schema` method can print appropriate messages to a log file, you must have previously specified a log handle (that is, a reference to a file object) by calling the `setLogHandle` method. Otherwise, the method sends these messages to `STDOUT`.

Example Usage

The following code fragment illustrates a typical call to the `schema` method using the *%typicalParams* hash parameter:

```
# Set up the hash list to contain the domain and database ID.
my %typicalParameters = (
                            domain => "NCOMS",
                            dbid => "NCIM"
                            );
```

```
# Call the schema method passing to it the previously set up hash list.
# In this case, the schema method knows that the name of the schema
# resides in a database log-ins configuration file. The schema method
# returns the name of the schema being used to the $schema variable.
#

my $schema = NCP::DBI_Factory::schema(%typicalParameters);
```

Consider the following entry in a `DbLogins.cfg` file. The previous call to the `schema` method would return a schema name of `ncim` (`m_schema`), which is associated with the database whose logical name is identified by the string `NCIM` (`m_DbId`).

```
insert into config.dbserver
(
    m_DbId,
    m_Server,
    m_DbName,
    m_Schema,
    m_Hostname,
    m_Username,
    m_Password,
    m_PortNum,
    m_EncryptedPwd
)
values
(
    "NCIM",  -- Logical name for this connection (don't change it)
    "MySQL",
    "ncim",
    "ncim",
    "localhost",
    "ncim",
    "ncim",
    3306,
    0
);
```

Returns

Upon completion, the `schema` method returns the name of the schema associated with the specified NCIM topology database.

See Also

- "createDbHandle" on page 137

setLogHandle

The `setLogHandle` method passes in the specified log handle associated with an opened file to which the `NCP::DBI_Factory` module methods can write messages.

Method Synopsis

`NCP::DBI_Factory::setLogHandle(`*`$filehandle`*`)`

Parameters

$filehandle
Specifies a reference to a file handle (for example, `IO::File`) that points to an opened file to which messages can be written.

Description

The `setLogHandle` method passes in the log handle specified in the *$filehandle* parameter to an internal utility method called by the `NCP::DBI_Factory` module methods. This handle is associated with an opened file to which this internal utility method writes messages. In effect, this opened file serves as a log file that can contain debug, critical, informational, and warning type messages associated with the execution of the `NCP::DBI_Factory` module methods.

If you do not call the `setLogHandle` method, the internal utility method writes these messages to `STDOUT`.

To control the level of message reporting, call the `setLogLevel` method and specify the desired log level.

Example Usage

The following code example shows a call to the `setLogHandle` method so that messages get logged to an open file (whose associated file handle is specified in the *$logFile* local variable) rather than to STDOUT. The code example also shows a call to the `setLogLevel` method that specifies the logging of messages at the `warn` and `critical` levels.

```
.
.
.
my $logName = "$logdir/checkPing.$domainName.log";

    my $logFile = new IO::File;
    $logFile->open(">$logName") or die "Could not open log file $logName\n";
    NCP::DBI_Factory::setLogHandle($logFile);
.
.
.

NCP::DBI_Factory::setLogLevel("warn");
```

Returns

Upon completion, the `setLogHandle` method returns no data.

See Also

- "setLogLevel"

setLogLevel

The `setLogLevel` method sets the log level for error and message reporting.

Method Synopsis

`NCP::DBI_Factory::setLogLevel(`*`$loglevel`*`)`

Parameters

$loglevel

Specifies the log level to set. The following are the valid options described in ascending order:

- `debug` — Specifies a log level in which all messages are logged.
- `info` — Specifies a log level in which informational, warning, and critical messages are logged.

- `warn` — Specifies a log level in which warning and critical messages are logged.
- `critical` — Specifies a log level in which only critical messages are logged.

Description

The `setLogLevel` method sets the log level to the option specified in the *$loglevel* parameter. The default is `debug` level. If set to a higher level, only messages with an equal or higher level will be logged. For example, at level `warn`, messages of level `info` and level `debug` will not be logged.

By default, the `NCP::DBI_Factory` module methods log messages to STDOUT. If you specify a log handle to the `setLogHandle` method, the `NCP::DBI_Factory` module methods log messages to the opened file associated with this log handle.

The `setLogLevel` method logs an appropriate message (either to STDOUT or to an opened file) if you specify an invalid log level.

Example Usage

The following code example illustrates a call to the `setLogLevel` method that specifies the logging of messages at the `warn` and `critical` levels. The code example also shows a call to the `setLogHandle` method so that these messages get logged to an open file (stored in the *$logFile* local variable) rather than to STDOUT:

```
.
.
.
my $logName = "$logdir/checkPing.$domainName.log";

    my $logFile = new IO::File;
    $logFile->open(">$logName") or die "Could not open log file $logName\n";
    NCP::DBI_Factory::setLogHandle($logFile);
.
.
.
NCP::DBI_Factory::setLogLevel("warn");
```

Returns

Upon completion, the `setLogLevel` method returns no data.

See Also

- "setLogHandle" on page 154

tables

The `tables` method returns a sorted array of table and view names for the current schema.

Method Synopsis

NCP::DBI_Factory::tables(*%dbhschema*)

NCP::DBI_Factory::tables(*%typicalParameters*)

Parameters

%dbhschema

Specifies a hash that contains the following keys:

- dbh — Specifies an existing DBI handle returned in a previous call to the createDbHandle method.
- schema — Specifies the schema that contains the tables of interest. Typically, this schema name is obtained in a call to the schema method.

%typicalParameters

Specifies the same hash parameter accepted by the createDbHandle method.

Description

The tables method returns a sorted array of table and view names for the current schema.

If you specify the *%dbhschema* hash, the tables method:

- Uses the dbh key/value pair to identify the existing DBI handle returned in a previous call to the createDbHandle method. This DBI handle provides the context for connecting to the specified NCIM topology database.
- Uses the current schema specified in the schema key/value pair to obtain the tables of interest.
- Returns a sorted array of the table and view names for all tables associated with the current schema.

If you specify the *%typicalParameters* hash, the tables method:

- Creates a new DBI handle. This DBI handle provides the context for connecting to the specified NCIM topology database.
- Uses the dbid key/value pair to identify the current schema. For example, if the dbid key/value pair is dbid => "NCIM", then the current schema might be called ncim.
- Uses the current schema identified in the dbid key/value pair to obtain the tables of interest.
- Returns a sorted array of the table and view names for all tables associated with the current schema.

Example Usage

The following code example illustrates a typical call to the tables method using the *%dbhschema* hash parameter. The code fragment also shows a call to the describeTable method:

```
# List the tables in schema $schema without creating a new DBI handle.
        my @tableList = NCP::DBI_Factory::tables(dbh => $dbh,
                                                  schema => $schema);
    foreach my $table (@tableList)
    {
        my @fields = NCP::DBI_Factory::describeTable(
                                              $table,
                                              dbh => $dbh,
                                              schema => $schema);
    }
```

Returns

Upon completion, the `tables` method returns a sorted array of table and view names for the current schema.

See Also

- "createDbHandle" on page 137
- "describeTable" on page 142
- "schema" on page 153

timeStamp

The `timeStamp` method returns a timestamp in a format suitable for addition to the NCIM topology database.

Method Synopsis

```
NCP::DBI_Factory::timeStamp([$unixtimestamp])
```

Parameters

$unixtimestamp
Specifies a UNIX timestamp. This is an optional parameter. If you do not specify this parameter, the `timeStamp` method uses the current timestamp on the local host.

Description

The `timeStamp` method converts the current timestamp on the local host (or the UNIX timestamp if specified in the *unixtimestamp* parameter) to the following format that is suitable for addition to the requested NCIM topology database:

```
YYYY-MM-DD HH:MM:SS
```

where:

- *YYYY* — Specifies the year.
- *MM* — Specifies the month.
- *DD* — Specifies the day.
- *HH* — Specifies the hour.
- *MM* — Specifies the minutes.
- *SS* — Specifies the seconds.

The `timeStamp` method adds leading zeroes to any of the previous fields whose values are less than 10.

Example Usage

The following code example illustrates a call to the `timeStamp` method, specifying the current timestamp on the local host:

```
.
.
.
my $currenttime
$currenttime = timestamp();
.
.
.
```

If the current timestamp on the local host is `June 6, 2010 5:39:45` EST, the `timeStamp` method converts it to the following format that is suitable for addition to the requested NCIM topology database:

```
2010-06-04-18:39:45
```

The following code example illustrates a call to the `timeStamp` method, specifying a UNIX timestamp:

```
my $currenttime
$currenttime = timestamp(1275694785);
```

The `timeStamp` method converts this UNIX timestamp to the following format that is suitable for addition to the requested database:

```
2010-06-04-18:39:45
```

Returns

Upon completion, the `currentTimeStamp` method returns the current timestamp on the local host (or the UNIX timestamp) in the following format:

```
YYYY-MM-DD HH:MM:SS
```

toUpper

The `toUpper` method returns a copy of a hash (a single row retrieved from an NCIM database table) with all field names converted to uppercase.

Method Synopsis

```
NCP::DBI_Factory::toUpper(%rowHashRef)
```

Parameters

%rowHashRef
Specifies a hash that is a single row retrieved from an NCIM database table. The field names in this row can be specified in mixed case, lowercase, or uppercase.

Description

The `toUpper` method takes the hash (a single row retrieved from an NCIM database table) specified in the *%rowHashRef* parameter and returns a copy of this hash with all field names converted to uppercase.

The reason for providing this method is to ensure consistency across databases. Different database implementations return field names in different formats (mixed case, uppercase, or lowercase). By always converting field names to uppercase, client scripts can be made database-server-independent. To do this, pass all returned rows through this method and perform any subsequent lookup operations with uppercase field names.

The `toUpper` method drops any undefined fields in the row to promote consistent behavior across the supported databases.

Notes

The `NCP::DBI_Factory` module supports DB2, Oracle, Informix, and MySql databases. In all of these databases, tables and field names are case insensitive with regard to SQL statements. However, note the following about field names in table rows returned by these databases:

- DB2 and Oracle — Return field names in uppercase.
- Informix — Returns field names in lowercase.
- MySql — Returns field names in mixed case.

Example Usage

The following code example makes calls to methods defined in the Perl DBI module to prepare, execute, and fetch a select statement:

```
my $statement = $dbh->prepare( $selectQuery );
$statement->execute();
my $results = $statement->fetchall_arrayref({});
```

The following list provides a line-by-line explanation of the previous code example:

- The first line calls the `prepare` method to prepare the select statement specified in *$selectQuery* for later execution by the database engine. The `prepare` method returns a reference to a statement handle object in *$statement*.
- The second line uses the statement handle object returned in *$statement* to get attributes of the select statement and then invokes the `execute` method to process the prepared statement.
- The third line calls the `fetchall_arrayref` method to fetch all the data returned from the previously prepared and executed select statement. The `fetchall_arrayref` method returns to *$results* a reference to an array that contains one reference per row.

The following code example calls the `toUpper` method to convert all field names to upper case:

```
foreach my $row (@$results)
    {
        $row = NCP::DBI_Factory::toUpper($row);
    }
```

The following list provides a line-by-line explanation of the previous code example:

- The first line sets up a `foreach` loop that iterates through the *@$results* array.
- For each field name in the table row, call the `toUpper` method to covert the name to uppercase.

The following code example shows that fields can be safely extracted using uppercase field names:

```
foreach my $row (@$results)
    {
        my $entityId = $row->{ENTITYID};
    }
```

Returns

Upon completion, the `toUpper` method returns a copy of a hash (a single row retrieved from a database table) with all field names converted to upper case.

NCP::Domain Reference

The `NCP::Domain` module provides an interface to perform operations on the Network Connectivity and Inventory Model (NCIM) topology database that resides in a single Network Manager domain.

The `NCP::Domain` module provides a constructor that creates a new `NCP::Domain` object that you use to call methods that perform these tasks:

- Create an entry in the domainMgr table for this domain if one does not already exist
- Create a new domain that is a copy of an existing domain
- Remove all references to the specified domain from the domainMgr table
- Retrieve the domainMgrId from the domainMgr table in the NCIM topology database that resides in the specified domain
- Return the domain name for the current domain
- Pass in a log handle associated with an opened file used for logging messages
- Set the log level for error and message reporting

Use of the methods that the `NCP::Domain` module provides assumes that you understand concepts related to the NCIM topology database.

See *IBM Tivoli Network Manager IP Edition Topology Database Reference* (SC27-2766-00) for information on NCIM topology databases.

The constructor and methods are described in reference (man) page format.

NCP::Domain module synopsis

The `NCP::Domain` module synopsis shows how to make calls to the constructor and domain operation methods that this module provides.

The comments provided in the synopsis serve as a quick reference as to the purpose of the constructor and domain operation methods. The reference (man) pages for the constructor and each method provide the details.

```
# Declare the module with the use directive.

use NCP::Domain;

# Call the NCP::Domain constructor. This call to the NCP::Domain
# constructor specifies the name of the Network Manager domain with
# the string "NEWDOMAIN". The NCP::Domain constructor returns
# to $domain a new NCP::Domain object.
#
# This call to the NCP::Domain constructor does not specify any
# database connection options.

my $domain = new NCP::Domain("NEWDOMAIN");

# Use the NCP::Domain object ($domain->) to directly invoke the
# create method to create an entry in the domainMgr table for the
# NEWDOMAIN domain.

$domain->create();

# Call the NCP::Domain constructor a second time. This call to
# the NCP::Domain constructor specifies the name of the Network
# Manager domain with a hash keyword/value pair of domain => "COPY".
# The NCP::Domain constructor returns to $copy a new NCP::Domain object.
```

```
#
# This call to the NCP::Domain constructor does not specify any
# database connection options.
my $copy = new NCP::Domain(domain => "COPY");

# Use the NCP::Domain object ($copy->) to directly invoke the clone
# method to create a new domain that is a copy of the existing
# domain (NEWDOMAIN).

$copy->clone("NEWDOMAIN");

# Call the NCP::Domain constructor a third time. This call to
# the NCP::Domain constructor specifies the name of the Network
# Manager domain with the string "OLD". The NCP::Domain constructor
# returns to $obsolete a new NCP::Domain object.
my $obsolete = new NCP::Domain("OLD");

# Use the NCP::Domain object ($obsolete->) to directly invoke
# the drop method to remove all references to the specified
# domain (OLD) from the domainMgr table.

$obsolete->drop();

# Use the NCP::Domain object ($domain->) created in the first
# call to the  NCP::Domain constructor to directly invoke the id
# method to retrieve the domain manager ID for this domain.

$domain->id();

# Use the NCP::Domain object ($domain->) created in the first
# call to the NCP::Domain constructor to directly invoke the name
# method to retrieve the domain name for this domain.

$domain->name();
```

See Also

- "NCP::Domain Constructor"
- "clone" on page 164
- "create" on page 164
- "drop" on page 166
- "id" on page 168
- "name" on page 169
- "setLogHandle" on page 171
- "setLogLevel" on page 172

NCP::Domain Constructor

The `NCP::Domain` constructor creates a blessed `NCP::Domain` object for the specified Network Manager domain.

Constructor

new NCP::Domain(*$domainName*)

new NCP::Domain(*$domainName*, *%dbOptionsHash*)

new NCP::Domain(*%dbOptionsHash*)

Parameters

$domainName
Specifies the name of the Network Manager domain for which you want to

create a blessed domain instance object. In this case, the domain name is specified with plain text or plain text assigned to a variable. You can also specify the domain name using the explicit hash key "domain".

%dbOptionsHash
Specifies the hash that contains the database login options. One of these database login options is the domain name. More specifically, this hash takes the same database login options as the DBI_Factory::createDbHandle method.

Description

The NCP::Domain constructor creates a blessed NCP::Domain object for the specified Network Manager domain. Use the NCP::Domain object (for example, $domain->) to invoke the methods that the NCP::Domain module provides.

The NCP::Domain constructor provides a great deal of flexibility on how you obtain the database login options for the *%dbOptionsHash* parameter. For example, you can call the NCP::DBI_Factory::extractCmdLineOptions method to ensure that the database login options specified on the command line are provided in a common format. The return from the NCP::DBI_Factory::extractCmdLineOptions method (a reference to a hash that contains the extracted database login options and values in key/value format) is passed to the *%dbOptionsHash* parameter.

Notes

Connection to the NCIM topology database that resides in this domain will be attempted only when required.

To ensure that the NCP::Domain constructor can print appropriate messages to a log file, you must have previously specified a log handle (that is, a reference to a file object) by calling the setLogHandle method. Otherwise, the constructor sends these messages to STDOUT.

Example Usage

The following code fragment illustrates a typical call to the NCP::Domain constructor:

```
my $domain = new NCP::Domain("NEWDOMAIN");
```

Returns

Upon completion, the NCP::Domain constructor returns a new NCP::Domain object.

See Also

- “createDbHandle” on page 137
- “extractCmdLineOptions” on page 145
- “setLogHandle” on page 171

clone

The `clone` method creates a new domain that is a copy of an existing domain.

Method Synopsis

`clone(`*`$domain`*`)`

Parameters

$domain
: Specifies the name of an existing Network Manager domain for which you want to create a copy. You created an instance of this domain by calling the `NCP::Domain` constructor.

Description

The `clone` method creates a new domain that is a copy of an existing domain.

Notes

The `clone` method assumes that the $NCHOMEenvironment variable is set. Otherwise, the clone method will not be able to copy the domain-specific configuration files from the existing domain.

To ensure that the `clone` method can print appropriate error and other messages to a log file, you must have previously specified a log handle (that is, a reference to a file object) by calling the `setLogHandle` method. Otherwise, the method sends these messages to `STDOUT`.

Returns

Upon completion, the `clone` method does not return any data.

See Also

- "NCP::Domain Constructor" on page 162
- "setLogHandle" on page 171

create

The `create` method creates an entry in the domainMgr table for the specified domain if one does not already exist.

Method Synopsis

`create()`

Parameters

None

Description

The `create` method creates an entry in the domainMgr table for the specified domain if one does not already exist. In addition, the `create` method creates an entry in the domainSummary table for the specified domain if one does not already exist. The domain name was specified in a previous call to the `NCP::Domain` constructor.

The domainMgr table stores data on network domains. For the specified domain, the `create` method inserts a row in the domainMgr table with values for the following table columns:

- domainName
- creationTime
- lastUpdated
- managerName
- webtopDataSource
- domainHost
- domainPort
- description

The domainMgr table contains an auto-increment column called domainMgrId, which the `create` method uses to automatically increment the field.

The domainSummary table stores statistical data on the specified domain. For the specified domain, the `create` method inserts a row in the domainSummary table with values for the following table columns:

- domainMgrId
- createTime
- changeTime

The `create` method logs appropriate error messages to a log file or `STDOUT` if it fails to insert an entry in the domainMgr table or the domainSummary table for the specified domain.

The `create` method permanently commits the insert row operations in the domainMgr and domainSummary tables to the database. Thus, it is not necessary for the application to call the Perl DBI module `commit` method.

Notes

You invoke the `create` method on the `NCP::Domain` object returned in a previous call to the `NCP::Domain` constructor. For example:

```
.
.
.
my $domain = new NCP::Domain($scriptOptions->{domain}, %$ncimArgs);
.
.
.
$domain->create();
```

To ensure that the `create` method can print appropriate error and other messages to a log file, you must have previously specified a log handle (that is, a reference to a file object) by calling the `setLogHandle` method. Otherwise, the method sends these messages to `STDOUT`.

Network Manager applications often use the `NCP::Domain` module methods in conjunction with the methods that the `NCP::DBI_Factory` module provides.

Example Usage

The following code shows a call to the `NCP::Domain` constructor, which returns the newly created `NCP::Domain` object to the *$domain* variable. The name of the domain is specified in *$domainName* in the call to the `NCP::Domain` constructor. The `%$ncimArgs` hash reference contains the command line arguments (database login options and values) in key/value format returned in a previous call to the `NCP::DBI_Factory::extractcmdLineOptions` method.

```
.
.
.
my $domain = new NCP::Domain($domainName, %$ncimArgs);
.
.
.
```

The following code shows that the `create` method is invoked on the `NCP::Domain` object (`$domain->`). The `create` method creates entries in the domainMgr and domainSummary tables for the domain specified in *$domainName*:

```
.
.
.
$domain->create();
.
.
.
```

Returns

Upon completion, the `create` method does not return any data.

See Also

- "NCP::Domain Constructor" on page 162
- "setLogHandle" on page 171
- "NCP::DBI_Factory module reference" on page 135

drop

The `drop` method removes all references to the specified domain from the domainMgr table.

Method Synopsis

`drop(`*`$domain`*`)`

Parameters

$domain
Specifies the name of an existing Network Manager domain for which you want to delete its associated entry from the domainMgr table. You created an instance of this domain by calling the `NCP::Domain` constructor.

Description

The `drop` method removes all references to the specified domain from the domainMgr table. This action effectively deletes any entities in the NCIM topology database for the specified domain. The `drop` method does not remove any configuration files associated with the specified domain.

Notes

You invoke the drop method on the NCP::Domain object returned in a previous call to the NCP::Domain constructor. For example:

```
.
.
.
my $domain = new NCP::Domain($scriptOptions->{domain}, %$ncimArgs);
.
.
.
$domain->drop(%$scriptOptions);
```

To ensure that the drop method can print appropriate error and other messages to a log file, you must have previously specified a log handle (that is, a reference to a file object) by calling the setLogHandle method. Otherwise, the method sends these messages to STDOUT.

Network Manager applications often use the NCP::Domain module methods in conjunction with the methods that the NCP::DBI_Factory module provides.

Example Usage

The example that illustrates a call to the drop method is divided into the following sections:

- Create a new NCP::Domain object
- Call drop to remove all references to the specified domain

Create a new NCP::Domain object

The following code shows a call to the NCP::Domain constructor, which returns a new NCP::Domain object to the *$domain* variable. The name of the domain is specified in *$domainName* in the call to the NCP::Domain constructor:

```
.
.
.
my $domain = new NCP::Domain($domainName, %$ncimArgs);
.
.
.
```

Call drop to remove all references to the specified domain

The following code shows an invocation of the drop method on the NCP::Domain object ($domain->):

```
.
.
.
$domain->drop(%$scriptOptions);
.
.
.
```

Returns

Upon completion, the drop method does not return any data.

See Also

- "NCP::Domain Constructor" on page 162
- "setLogHandle" on page 171
- "NCP::DBI_Factory module reference" on page 135

id

The `id` method retrieves the domainMgrId from the domainMgr table in the NCIM topology database that resides in the specified domain.

Method Synopsis

```
id()
```

Parameters

None

Description

The `id` method retrieves the domainMgrId from the domainMgr table in the NCIM topology database that resides in this domain.

Notes

You invoke the `id` method on the `NCP::Domain` object returned in a previous call to the `NCP::Domain` constructor. For example:

```
.
.
.
my $domain = new NCP::Domain($domainName, %$ncimArgs);
.
.
.
my $domainMgrId = $domain->id();
.
.
.
```

To ensure that the `id` method can print appropriate error and other messages to a log file, you must have previously specified a log handle (that is, a reference to a file object) by calling the `setLogHandle` method. Otherwise, the method sends these messages to `STDOUT`.

Example Usage

The example that illustrates a call to the `id` method is divided into the following sections:

- Create a new `NCP::Domain` object
- Call `id` to return the domainMgrId

Create a new NCP::Domain object

The following code shows a call to the `NCP::Domain` constructor, which returns a new `NCP::Domain` object to the *$domain* variable. The name of the domain is specified in *$domainName* in the call to the `NCP::Domain` constructor:

```
.
.
.
my $domain = new NCP::Domain($domainName, %$ncimArgs);
.
.
.
```

Call `id` to return the name of the domain

The following code shows an invocation of the `id` method on the `NCP::Domain` object (`$domain->`). The `id` method returns the domainMgrId to the *$domainMgrId* variable. Note that *$domainMgrId* is used in the call to the `dropPollPolicies` method.

```
.
.
.
my $domainMgrId = $domain->id();
.
.
.

dropPollPolicies($ncmonitor, $domainMgrId);
.
.
.
```

Returns

Upon completion, the `id` method returns the domainMgrId.

See Also

- "NCP::Domain Constructor" on page 162
- "setLogHandle" on page 171

name

The `name` method returns the name of the domain.

Method Synopsis

`name()`

Parameters

None

Description

The `name` method returns the name of the domain.

Notes

You invoke the `name` method on the `NCP::Domain` object returned in a previous call to the `NCP::Domain` constructor. For example:

```
.
.
.
my $domain = new NCP::Domain($domainName, %$ncimArgs);
.
```

```
.
.
my $domainName = $domain->name();
.
.
.
```

Example Usage

The example that illustrates a call to the `name` method is divided into the following sections:
- Create a new `NCP::Domain` object
- Call `name` to return the name of the domain

Create a new NCP::Domain object

The following code shows a call to the `NCP::Domain` constructor, which returns a new `NCP::Domain` object to the *$domain* variable. The name of the domain is specified in *$domainName* in the call to the `NCP::Domain` constructor:

```
.
.
.
my $domain = new NCP::Domain($domainName, %$ncimArgs);
.
.
.
```

Call `name` to return the name of the domain

The following code shows an invocation of the `name` method on the `NCP::Domain` object (`$domain->`). The `name` method returns the name of the domain to the *$domainName* variable. Note that *$domainName* is used in the call to the `createDbHandle` method.

```
.
.
.
my $domainName = $domain->name();
.
.
.

    my $ncmonitor = NCP::DBI_Factory::createDbHandle(domain => $domainName,
                                                     dbid => "NCMONITOR",
                                                     %$ncmonitorArgs);
.
.
.
```

Returns

Upon completion, the `name` method returns the name of the domain.

See Also
- "NCP::Domain Constructor" on page 162
- "createDbHandle" on page 137

setLogHandle

The `setLogHandle` method passes in a log handle associated with an opened file to the `NCP::Domain` module methods. These methods can then write messages to this opened file.

Method Synopsis

```
setLogHandle($filehandle)
```

Parameters

$filehandle
: Specifies a reference to a file handle (for example, `IO::File`) that points to an opened file to which messages can be written.

Description

The `setLogHandle` method passes in the log handle specified in the *$filehandle* parameter to an internal utility method called by the `NCP::Domain` module methods. This handle is associated with an opened file to which this internal utility method writes messages. In effect, this opened file serves as a log file that contains critical, informational, and warning type messages associated with the execution of the `NCP::Domain` module methods.

If you do not call the `setLogHandle` method, the internal utility method writes these messages to `STDOUT`.

Notes

You invoke the `setLogHandle` method on the `NCP::Domain` object returned in a previous call to the `NCP::Domain` constructor. For example:

```
.
.
.
my $domain = new NCP::Domain($domainName, %$ncimArgs);
.
.
.
$domain->setLogHandle($logFile);
.
.
.
```

Example Usage

The example that illustrates a call to the `setLogHandle` method is divided into the following sections:
- Create a new `NCP::Domain` object
- Call `setLogHandle` to log messages to a specified open file

Create a new NCP::Domain object

The following code shows a call to the `NCP::Domain` constructor, which returns a new `NCP::Domain` object to the *$domain* variable. The name of the domain is specified in *$domainName* in the call to the `NCP::Domain` constructor:

```
.
.
.
```

```
my $domain = new NCP::Domain($domainName, %$ncimArgs);
.
.
.
```

Call `setLogHandle` to log messages to a specified open file

The following code shows an invocation of the `setLogHandle` method on the `NCP::Domain` object (`$domain->`). The call to the `setLogHandle` method ensures that any messages get logged to an open file (whose associated file handle is specified in the *$logFile* local variable) rather than to STDOUT:

```
.
.
.
my $logName = "$logdir/checkDomain.$domainName.log";

    my $logFile = new IO::File;
    $logFile->open(">$logName") or die "Could not open log file $logName\n";
    $domain->setLogHandle($logFile);
.
.
.
```

Returns

Upon completion, the `setLogHandle` method returns no data.

See Also

- "NCP::Domain Constructor" on page 162

setLogLevel

The `setLogLevel` method sets the log level for error and message reporting.

Method Synopsis

`setLogLevel(`*`$loglevel`*`)`

Parameters

$loglevel

Specifies the log level to set. The following are the valid options described in ascending order:

- `debug` — Specifies a log level in which all messages are logged.
- `info` — Specifies a log level in which informational, warning, and critical messages are logged.
- `warn` — Specifies a log level in which warning and critical messages are logged.
- `critical` — Specifies a log level in which only critical messages are logged.

Description

The `setLogLevel` method sets the log level to the option specified in the *$loglevel* parameter. The default is `debug` level. If set to a higher level, only messages with an equal or higher level will be logged. For example, at level `warn`, messages of level `info` and level `debug` will not be logged.

By default, the `NCP::Domain` module methods log messages to STDOUT. If you specify a log handle to the `setLogHandle` method, the `NCP::Domain` module methods log messages to the opened file associated with this log handle.

The `setLogLevel` method logs an appropriate message (either to STDOUT or to an opened file) if you specify an invalid log level.

Notes

You invoke the `setLogLevel` method on the `NCP::Domain` object returned in a previous call to the `NCP::Domain` constructor. For example:

```
.
.
.
my $domain = new NCP::Domain($domainName, %$ncimArgs);
.
.
.
$domain->setLogLevel("warn");
.
.
.
```

Example Usage

The example that illustrates a call to the `setLogLevel` method is divided into the following sections:
- Create a new `NCP::Domain` object
- Call `setLogHandle` to log messages to a specified open file
- Call `setLogLevel` to set the log level

Create a new NCP::Domain object

The following code shows a call to the `NCP::Domain` constructor, which returns a new `NCP::Domain` object to the *$domain* variable. The name of the domain is specified in *$domainName* in the call to the `NCP::Domain` constructor:

```
.
.
.
my $domain = new NCP::Domain($domainName, %$ncimArgs);
.
.
.
```

Call `setLogHandle` to log messages to a specified open file

The following code shows an invocation of the `setLogHandle` method on the `NCP::Domain` object ($domain->). The call to the `setLogHandle` method ensures that any messages get logged to an open file (stored in the *$logFile* local variable) rather than to STDOUT:

```
.
.
.
my $logName = "$logdir/checkDomain.$domainName.log";

    my $logFile = new IO::File;
    $logFile->open(">$logName") or die "Could not open log file $logName\n";
    $domain->setLogHandle($logFile);
.
.
.
```

Call `setLogLevel` to set the log level

The following code shows that the `setLogLevel` method is invoked on the `NCP::Domain` object (`$domain->`). The call to `setLogLevel` specifies the logging of messages at the `warn` and `critical` levels.

```
.
.
.
$domain->setLogLevel("warn");
.
.
.
```

Returns

Upon completion, the `setLogLevel` method returns no data.

See Also

- "NCP::Domain Constructor" on page 162
- "setLogHandle" on page 171

Appendix C. Network Manager glossary

Use this information to understand terminology relevant to the Network Manager product.

The following list provides explanations for Network Manager terminology.

AOC files
: Files used by the Active Object Class manager, `ncp_class` to classify network devices following a discovery. Device classification is defined in AOC files by using a set of filters on the object ID and other device MIB parameters.

active object class (AOC)
: An element in the predefined hierarchical topology of network devices used by the Active Object Class manager, `ncp_class`, to classify discovered devices following a discovery.

agent
: See, discovery agent.

class hierarchy
: Predefined hierarchical topology of network devices used by the Active Object Class manager, `ncp_class`, to classify discovered devices following a discovery.

configuration files
: Each Network Manager process has one or more configuration files used to control process behaviour by setting values in the process databases. Configuration files can also be made domain-specific.

discovery agent
: Piece of code that runs during a discovery and retrieves detailed information from discovered devices.

Discovery Configuration GUI
: GUI used to configure discovery parameters.

Discovery engine (ncp_disco)
: Network Manager process that performs network discovery.

discovery phase
: A network discovery is divided into four phases: Interrogating devices, Resolving addresses, Downloading connections, and Correlating connectivity.

discovery seed
: One or more devices from which the discovery starts.

discovery scope
: The boundaries of a discovery, expressed as one or more subnets and netmasks.

Discovery Status GUI
: GUI used to launch and monitor a running discovery.

discovery stitcher
: Piece of code used during the discovery process. There are various discovery stitchers, and they can be grouped into two types: data collection stitchers, which transfer data between databases during the data collection

phases of a discovery, and data processing stitchers, which build the network topology during the data processing phase.

domain
See, network domain.

entity
A topology database concept. All devices and device components discovered by Network Manager are entities. Also device collections such as VPNs and VLANs, as well as pieces of topology that form a complex connection, are entities.

event enrichment
The process of adding topology information to the event.

Event Gateway (ncp_g_event)
Network Manager process that performs event enrichment.

Event Gateway stitcher
Stitchers that perform topology lookup as part of the event enrichment process.

failover
In your Network Manager environment, a failover architecture can be used to configure your system for high availability, minimizing the impact of computer or network failure.

Failover plug-in
Receives Network Manager health check events from the Event Gateway and passes these events to the Virtual Domain process, which decides whether or not to initiate failover based on the event.

Fault Finding View
Composite GUI view consisting of an **Active Event List (AEL)** portlet above and a Network Hop View portlet below. Use the Fault Finding View to monitor network events.

full discovery
A discovery run with a large scope, intended to discover all of the network devices that you want to manage. Full discoveries are usually just called discoveries, unless they are being contrasted with partial discoveries. See also, partial discovery.

message broker
Component that manages communication between Network Manager processes. The message broker used byNetwork Manager is called Really Small Message Broker. To ensure correct operation of Network Manager, Really Small Message Broker must be running at all times.

NCIM database
Relational database that stores topology data, as well as administrative data such as data associated with poll policies and definitions, and performance data from devices.

ncp_disco
See, Discovery engine.

ncp_g_event
See, Event Gateway.

ncp_model
See, Topology manager.

ncp_poller
See, Polling engine.

network domain
A collection of network entities to be discovered and managed. A single Network Manager installation can manage multiple network domains.

Network Health View
Composite GUI view consisting of a Network Views portlet above and an **Active Event List (AEL)** portlet below. Use the Network Health View to display events on network devices.

Network Hop View
Network visualization GUI. Use the Network Hop View to search the network for a specific device and display a specified network device. You can also use the Network Hop View as a starting point for network troubleshooting. Formerly known as the Hop View.

Network Polling GUI
Administrator GUI. Enables definition of poll policies and poll definitions.

Network Views
Network visualization GUI that shows hierarchically organized views of a discovered network. Use the Network Views to view the results of a discovery and to troubleshoot network problems.

OQL databases
Network Manager processes store configuration, management and operational information in OQL databases.

OQL language
Version of the Structured Query Language (SQL) that has been designed for use in Network Manager. Network Manager processes create and interact with their databases using OQL.

partial discovery
A subsequent rediscovery of a section of the previously discovered network. The section of the network is usually defined using a discovery scope consisting of either an address range, a single device, or a group of devices. A partial discovery relies on the results of the last full discovery, and can only be run if the Discovery engine, `ncp_disco`, has not been stopped since the last full discovery. See also, full discovery.

Path Views
Network visualization GUI that displays devices and links that make up a network path between two selected devices. Create new path views or change existing path views to help network operators visualize network paths.

performance data
Performance data can be gathered using performance reports. These reports allow you to view any historical performance data that has been collected by the monitoring system for diagnostic purposes.

Polling engine (ncp_poller)
Network Manager process that polls target devices and interfaces. The Polling engine also collects performance data from polled devices.

poll definition
Defines how to poll a network device or interface and further filter the target devices or interfaces.

poll policy
: Defines which devices to poll. Also defines other attributes of a poll such as poll frequency.

Probe for Tivoli Netcool/OMNIbus (nco_p_ncpmonitor)
: Acquires and processes the events that are generated by Network Manager polls and processes, and forwards these events to the ObjectServer.

RCA plug-in
: Based on data in the event and based on the discovered topology, attempts to identify events that are caused by or cause other events using rules coded in RCA stitchers.

RCA stitcher
: Stitchers that process a trigger event as it passes through the RCA plug-in.

root-cause analysis (RCA)
: The process of determining the root cause of one or more device alerts.

SNMP MIB Browser
: GUI that retrieves MIB variable information from network devices to support diagnosis of network problems.

SNMP MIB Grapher
: GUI that displays a real-time graph of MIB variables for a device and usse the graph for fault analysis and resolution of network problems.

stitcher
: Code used in the following processes: discovery, event enrichment, and root-cause analysis. See also, discovery stitcher, Event Gateway stitcher, and RCA stitcher.

Structure Browser
: GUI that enables you to investigate the health of device components in order to isolate faults within a network device.

Topology Manager (ncp_model)
: Stores the topology data following a discovery and sends the topology data to the NCIM topology database where it can be queried using SQL.

WebTools
: Specialized data retrieval tools that retrieve data from network devices and can be launched from the network visualization GUIs, Network Views and Network Hop View, or by specifying a URL in a web browser.

Notices

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Trademarks

The terms in Table 2 are trademarks of International Business Machines Corporation in the United States, other countries, or both:

Table 2. IBM trademarks

AIX	iSeries	RDN
ClearQuest	Lotus	SecureWay
Cognos	Netcool	solidDB
Current	NetView	System z
DB2	Notes	Tivoli
developerWorks	OMEGAMON	WebSphere
Enterprise Storage Server	PowerVM	z/OS
IBM	PR/SM	z/VM
Informix	pSeries	zSeries

Intel, Intel logo, Intel Inside, Intel Inside logo, Intel Centrino, Intel Centrino logo, Celeron, Intel Xeon, Intel SpeedStep, Itanium, and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Java™ and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Index

A

accessibility ix
audience v

C

constructors
  NCP::Domain 162
  RIV::Agent 73
  RIV::App 95
  RIV::OQL 97
  RIV::Param 106
  RIV::Record 113
  RIV::RecordCache 119
  RIV::SnmpAccess 123
conventions, typeface x

E

education
  see Tivoli technical training x
environment variables, notation x

F

functions
  RIV::GetInput 62
  RIV::GetResult 63
  RIV::InputFilter 65
  RIV::InputQueueLength 66
  RIV::IsIpNotLoopBackOrMulticast 66
  RIV::IsIpv4Valid 68
  RIV::IsIpv6Valid 68
  RIV::IsIpValid 67
  RIV::ReadDir 69
  RIV::RivDebug 70
  RIV::RivError 70
  RIV::RivMessage 71

G

glossary 175

M

manuals vi
methods
  AddLocalNeighbour 114
  AddLocalNeighbourTag 115
  AddRemoteNeighbour 115
  AddRemoteNeighbourTag 116
  ASN1ToOid 123, 132
  AsyncSnmpGet 124
  AsyncSnmpGetBulk 125
  AsyncSnmpGetNext 127
  CacheRecord 120
  clone 164
  create 164

methods *(continued)*
  CreateDB 98
  createDbHandle 137
  CreateTable 98
  Delete 99
  describeTable 142
  DomainName 110, 111
  drop 166
  execute_insert_auto_inc 144
  extractCmdLineOptions 145
  extractHashRefOptions 147
  GetDNSAllIpAddrs 74
  GetDNSAllNames 74
  GetDNSFirstIpAddr 75
  GetDNSFirstName 76
  GetIpArp 77
  GetLocalNeighbours 117
  GetMacArp 77
  GetMibHash 128
  GetMultTelnet 78
  GetPingIP 79
  GetPingList 80
  GetPingSubnet 80
  GetRecord 120
  GetRecords 121
  GetRemoteNeighbours 117
  GetTelnet 81
  GetTelnetCols 82
  GetTraceRoute 83
  id 168
  Insert 100
  insert_auto_inc_row 149
  insert_row 150
  LockThreads 83
  name 169
  OidToASN1 129
  PingIP 84
  PingList 85
  PingSubnet 85
  prepare_insert_auto_inc 152
  Print 102, 118
  schema 153
  Select 102
  Send 104
  SendNeToDisco 86
  SendNEToNextPhase 87
  setLogHandle 154, 171
  setLogLevel 155, 172
  SnmpGet 91, 129
  SnmpGetBulk 92, 130
  SnmpGetNext 93, 131
  tables 156
  timeStamp 158
  toUpper 159
  UnLockThreads 94
  Update 104
  Usage 112
module synopsis
  NCP::DBI::Factory 135
  NCP::Domain 161
  RIV 53

module synopsis *(continued)*
  RIV::Agent 72
  RIV::App 94
  RIV::OQL 96
  RIV::Param 106
  RIV::Record 113
  RIV::RecordCache 118
  RIV::SnmpAccess 122

N

Network Manager glossary 175

O

Object Query Language 35
online publications vi
OQL 35
ordering publications vi

P

publications vi

S

support information x

T

Tivoli software information center vi
Tivoli technical training x
training, Tivoli technical x
typeface conventions x

V

variables
  DebugLevel 56, 58
  MaxAsyncConcurrrent 128
variables, notation for x
virtual methods
  AddSubject 54
  AddTimer 55
  DecryptPassword 56
  EncryptPassword 57
  Latency 57
  PostInput 59
  PublishMessage 59, 60
  RetryLimit 61

W

What this publication contains v

Printed in the Republic of Ireland

SC27-2769-01