



Cisco IOS XR Perl Scripting Toolkit

This chapter describes the Cisco IOS XR Perl Scripting Toolkit as an alternative method to the existing management methods that enables the router to be managed by a Perl script running on a separate machine. Management commands and data are sent to and from the router in the form of extensible markup language (XML) over either a Telnet or an SSH connection. The well-defined and consistent structure of XML, which is used for both commands and data, makes it easy to write scripts that can interactively manage the router, display information returned from the router in the format required, or manage multiple routers at once.

The following sections describe how to use the Cisco IOS XR Perl Scripting Toolkit:

- [Cisco IOS XR Perl Scripting Toolkit Concepts, page 15-130](#)
- [Security Implications for the Cisco IOS XR Perl Scripting Toolkit, page 15-130](#)
- [Prerequisites for Installing the Cisco IOS XR Perl Scripting Toolkit, page 15-130](#)
- [Installing the Cisco IOS XR Perl Scripting Toolkit, page 15-131](#)
- [Using the Cisco IOS XR Perl XML API in a Perl Script, page 15-132](#)
- [Handling Types of Errors for the Cisco IOS XR Perl XML API, page 15-132](#)
- [Starting a Management Session on a Router, page 15-132](#)
- [Closing a Management Session on a Router, page 15-134](#)
- [Sending an XML Request to the Router, page 15-134](#)
- [Using Response Objects, page 15-134](#)
- [Using the Error Objects, page 15-135](#)
- [Using the Configuration Services Methods, page 15-136](#)
- [Using the Cisco IOS XR Perl Data Object Interface, page 15-138](#)
- [Cisco IOS XR Perl Notification and Alarm API, page 15-146](#)
- [Examples of Using the Cisco IOS XR Perl XML API, page 15-150](#)

Cisco IOS XR Perl Scripting Toolkit Concepts

Table 15-1 describes the toolkit concepts. Some sample scripts are modified and show how to use the API in your own scripts.

Table 15-1 List of Concepts for the IOS XR Perl Scripting Toolkit

Concept	Definition
Cisco IOS XR Perl XML API ¹	Consists of the core of the toolkit and provides the ability to create management sessions, send management requests, and receive responses by using Perl objects and methods.
Cisco IOS XR Perl Data Object API	Allows management requests to be sent and responses received entirely using Perl objects and data structures without any knowledge of the underlying XML.
Cisco IOS XR Perl Notification/Alarm API	Allows a script to register for notifications (for example, alarms), on a management session and receive the notifications asynchronously as Perl objects.

1. API = application programming interface

Security Implications for the Cisco IOS XR Perl Scripting Toolkit

Similar to using the command-line interface (CLI) over a Telnet or Secured Shell (SSH) connection, all authentication and authorization are handled by authentication, authorization, and accounting (AAA) on the router. A script prompts you to enter a password at run time, which ensures that passwords never get stored on the client machine. Therefore, the security implications for using the toolkit are identical to the CLI over the same transport.

Prerequisites for Installing the Cisco IOS XR Perl Scripting Toolkit

To use the toolkit, you must have installed Perl version 5.6 on the client machine that runs UNIX and Linux. To use the SSH transport option, you must have the SSH client executable installed on the machine and in your path.

You need to install the following specific standard Perl modules to use the various functions:

- `XML::LibXML`—This module is essential for using the Perl XML API and requires that the `libxml2` library be installed on the system first, which must be the version that is compatible with the version of `XML::LibXML`. The toolkit is tested to work with `XML::LibXML` version 1.58 and `libxml2` version 2.6.6. If you are installing `libxml2` from a source, you must apply the included patch file before compiling.
- `Term::ReadKey` (optional but recommended)—This module reads passwords without displaying them on the screen.
- `Net::Telnet`—This module is needed if you are using the Telnet or SSH transport modules.

If one of the modules is not available in the current version, you are warned during the installation process. Before installing the toolkit, you should install the current versions of the modules. You can obtain all modules from the following location: <http://www.cpan.org/>

The following modules are not necessary for using the API, but are required to run some sample scripts:

- **XML::LibXSLT**—This module is needed for the sample scripts that use XSLT to produce HTML pages. The module also requires that the libxslt library be installed on the system first. The toolkit is tested to work with XML::LibXSLT version 1.57 and libxslt version 1.1.3.
- **Mail::Send**—This module is needed only for the notifications example script.

Installing the Cisco IOS XR Perl Scripting Toolkit

The Cisco IOS XR Perl Scripting Toolkit is distributed in a file named `Cisco-IOS_XR-Perl-Scripting-Toolkit-<version>.tar.gz`.

To install the Cisco IOS XR Perl Scripting Toolkit, perform the following steps:

Step 1 Extract the contents from the directory in which the file resides by entering the following command:

```
tar -f Cisco-IOS_XR-Perl-Scripting-Toolkit-<version>.tar.gz -xzc <destination>
```

Table 15-2 defines the parameters.

Table 15-2 Toolkit Installation Directory Parameters

Name	Description
<version>	Defines the version of the toolkit that you want to install, for example, version 1.0.
<destination>	Specifies the existing directory in which you want to create the toolkit installation directory. A directory called <code>Cisco-IOS_XR-Perl-Scripting-Toolkit-<version></code> is created within the <destination> directory along with the extracted contents.

Step 2 Use the **cd** command to change to the toolkit installation directory and enter the following command:

```
perl Makefile.PL
```

If the command gives a warning that one of the prerequisite modules is not found, download and install the applicable module from the Comprehensive Perl Archive Network (CPAN) before using the API.

Step 3 Use the **make** command to maintain a set of programs, as shown in the following example:

```
make
```

Step 4 Use the **make install** command, as shown in the following example:

```
make install
```

Ensure that you have the applicable permission requirements for the installation. You may need to have root privileges.

If you do not encounter any errors, the toolkit is installed successfully. The Perl modules are copied into the appropriate directory, and you can use your own Perl scripts.

Using the Cisco IOS XR Perl XML API in a Perl Script

To use the Cisco IOS XR Perl XML API in a Perl application, import the module by including the following statement at the top of the script:

```
use Cisco::IOS_XR;
```

If you are using the Data Object interface, you can specify extra import options in the statement. For more information about the objects, see [“Creating Data Objects” section on page 15-139](#).

Handling Types of Errors for the Cisco IOS XR Perl XML API

The following types of errors can occur when using the Cisco IOS XR Perl XML API:

- Errors returned from the router—Specify that the errors are produced during the processing of an XML request and are returned to you in an XML response document. For more information about how these errors are handled, see [“Using the Error Objects” section on page 15-135](#).
- Errors produced within the Perl XML API modules—Specify that the script cannot continue. The module causes the script to be terminated with the appropriate error message. If the script writer wants the script to handle these error types, the writer must write the die handlers (for example, enclose the call to the API function within an `eval{}` block).

Starting a Management Session on a Router

Before any requests are sent, a management session must be started on the router, which is done by creating a new object of type named `Cisco::IOS_XR`. The new object is used for all further requests during the session, and the session is ended when the object is destroyed. A `Cisco::IOS_XR` object is created by calling `Cisco::IOS_XR::new`.

[Table 15-3](#) lists the optional parameters specified as arguments.

Table 15-3 Argument Definitions

Name	Description
<i>use_command_line</i>	Controls whether or not the <code>new()</code> method parses the command-line options given when the script was invoked. If the value of the argument is true, which is the default, the command-line options specify or override any of the subsequent arguments and control debug and logging options. The value of 0 defines the value as false.
<i>interactive</i>	If the value of the argument is true, the script prompts you for the username and password if they have not been specified either in the script or on the command line. The <code>Term::ReadKey</code> module must be installed. The most secure way of using the toolkit is for the input not to be echoed to the screen, which avoids hard coding or any record of passwords having been used. The default value is false. For example, the script does not ask for user input. As a command-line option, the <code>interactive</code> argument does not take any arguments. You can specify <code>-interactive</code> to turn on the interactive mode.

Table 15-3 Argument Definitions (continued)

Name	Description
<i>transport</i>	Means by which the Perl application should connect to the router, which defaults to Telnet. If a different value is specified, the <code>new()</code> method searches for a package called <code>Cisco::IOS_XR::Transport::<transport_name></code> . If found, the Perl application uses that package to connect to the router.
<i>ssh_version</i>	If the chosen transport option is SSH and the SSH executable on your system supports SSH v2, specifies which version of SSH you want to use for the connection. The valid values are 1 and 2. If the SSH executable supports only version 1, an error is caused by specifying the <i>ssh_version</i> argument.
<i>host</i>	Specifies the name or IP address of the router to which to connect. The router console or auxiliary ports should not be used because they are likely to cause problems for the script when logging in and offer significantly lower performance than a management port.
<i>port</i>	Specifies the TCP port for the connection. The default value depends on the transport being used.
<i>username</i>	Specifies the username to log in to the router.
<i>password</i>	Specifies the corresponding password.
<i>connection_timeout</i>	Specifies the timeout value that is used to connect and log in to the session. If not specified, the default value is 5 seconds.
<i>response_timeout</i>	Specifies the timeout value that is used when waiting for a response to an XML request. If not specified, the default value is 10 seconds.
<i>prompt</i>	Specifies the prompt that is displayed on the router after a successful log in. The default is <code><host>#</code> .

The following example shows the arguments given using the standard Perl hash notation:

```
use Cisco::IOS_XR;
my $session = new Cisco::IOS_XR(transport => 'telnet',
                                host => 'router1',
                                port => 7000,
                                username => 'john',
                                password => 'smith',
                                connection_timeout => 3);
```

[Table 15-4](#) describes the additional command-line options that can be specified.

Table 15-4 Command-Line Options

Name	Description
debug	Turns on the specified debug type and can be repeated to turn on more than one type.
logging	Turns on the specified logging type and can be repeated to turn on more than one type.
log_file	Specifies the name of the log file to use.
telnet_input_log	Specifies the file used for the Telnet input log, if you are using Telnet.
telnet_dump_log	Specifies the file used for the Telnet dump log, if you are using Telnet.

To use the command-line options when invoking a script, use the `-option value` format (assuming the option has a value). The option name does not need to be given in full, but must be long enough to be distinguished from other options. The following example is displayed:

```
perl my_script.pl -host my_router -user john -interactive -debug xml
```

Closing a Management Session on a Router

When an object of type `Cisco::IOS_XR` is created, the transport connection to the router and any associated resources on the router are maintained until the object is destroyed and automatically cleaned. For most scripts, the process should occur automatically when the script ends.

If you want to close a particular session during the course of the script, use the `close()` method. You can perform an operation on a large set of routers sequentially, and not keep all sessions open for the duration of the script, as displayed in the following example:

```
my $session1 = new Cisco::IOS_XR(host => 'router1', ...);
#do some stuff
$session1->close;
my $session2 = new Cisco::IOS_XR(host => 'router2', ...);
# do some stuff
...
```

Sending an XML Request to the Router

Requests and responses pass between the client and router in the form of XML. Depending on whether the XML is stored in a string or file, you can construct an XML request that is sent to the router using either the `send_req` or `send_req_file` method. Some requests are sent without specifying any XML by using the configuration services methods, for example, `commit` and `lock` or the Data Object interface.

The following example shows how to send an XML request in the form of a string:

```
my $xml_req_string = '<?xml...><Request>...</Request>';
my $response = $session->send_req($xml_req_string);
```

The following example shows how to send a request stored in a file:

```
my $response = $session->send_req_file('request.xml');
```

Using Response Objects

Both of the `send_req` and `send_req_file` methods return a `Cisco::IOS_XR::Response` object, which contains the XML response returned by the router.



Note

Both send methods handle iterators in the background; so if a response consists of many parts, the response object returned is the result of merging them back together.

Retrieving the Response XML as a String

The following example shows how to use the `to_string` method:

```
$xml_response_string = $response->to_string;
```

Writing the Response XML Directly to a File

The following example shows how to use the `write_file` method by specifying the name of the file to be written:

```
$response->write_file('response.xml');
```

Retrieving the Data Object Model Tree Representation of the Response

The following example shows how to retrieve a Data Object Model (DOM) tree representation for the response:

```
my $document = $response->get_dom_tree;
```

You should be familiar with the DOM, which an XML document represents an object tree structure. For more information, see <http://www.w3.org/DOM/>



Note

The returned DOM tree type will be of type `XML::LibXML::Document`, because this is the form in which the response is held internally. The method is quick, because it does not perform extra parsing and should be used in preference to retrieving the string form of the XML and parsing it again (unless a different DOM library is used).

Determining if an Error Occurred While Processing a Request

The following example shows how to determine if an error occurred while processing a request:

```
my $error = $response->get_error;
if (defined($error)) {
    die $error;
}
```

If it is only of interest whether errors occurred but not what all the errors were, use the `get_error` method to return one error from the response. This returns an error object that represents the first error found or is undefined if none were found.

Retrieving a List of All Errors Found in the Response XML

The following example shows how to list all errors that occurred rather than just one by using the `get_errors` method:

```
my @errors = $response->get_errors;
```

The `get_errors` method returns an array of error objects that represents all errors that were found in the response XML. For more information, see “Using the Error Objects” section on page 15-135.

Using the Error Objects

Error objects are returned when calling the `get_error` and `get_errors` methods on a response object, and are used to represent an error encountered in an XML response. Table 15-5 lists the methods for the object.

Table 15-5 List of Methods for the Object

Method	Description
get_message	Returns the error message string that was found in the XML.
get_code	Returns the corresponding error code.
get_element	Returns the tag name of the XML element in which the error was found.
get_dom_node	Returns a reference to the element node in the response DOM ¹ tree.
to_string	Returns a string that contains the error message, code, and element name. If the error object is used in a scalar context, the method is used automatically to convert it to a string. The following example displays all information in an error: Error encountered in object ConfederationPeerASTable: 'XMLMDA' detected the 'warning' condition 'The XML request does not conform to the schema. A child element of the element on which this error appears includes a non-existent naming, filter, or value element. Please check the request against the schema.' Error code: 0x4368a000

1. DOM = Data Object Model

Using the Configuration Services Methods

Methods are provided to enable the standard configuration services operations to be performed without knowledge of the underlying XML. These are the operations that are usually performed at the start or end of a configuration session, such as locking the running configuration or saving the configuration to a file.

The `config_commit()` function takes the following optional arguments:

- *mode*
- *label*
- *comment*

Committing the Target Configuration

The arguments are specified in any order using the standard Perl hash notation, as shown in the following example:

```
$response = $session->config_commit(Label => 'Example1', Comment => 'Just an example');
```

A response object is returned from which any errors can be extracted, if desired. To retrieve the commit ID that was assigned to the commit upon success, you can call the `get_commit_id()` method on the response object, as shown in the following example:

```
$commit_id = $response->get_commit_id();
```

Locking and Unlocking the Running Configuration

The following example shows how to use the `config_lock` and `config_unlock` functions, which take no arguments:

```
$error = $session->config_lock;
$error = $session->config_unlock;
```

Loading a Configuration from a File

The following example shows how to contain a filename as an argument:

```
$error = $session->config_load(Filename => 'test_config.cfg');
```

Loading a Failed Configuration

The following example shows how to use the `config_load_failed` function, which takes no arguments:

```
$error = $session->config_load_failed;
```

Saving a Configuration to a File

The following example shows how to use two arguments for the `config_save()` function:

```
$error = $session->config_save(Filename => 'disk0:/my_config.cfg', Overwrite => 'true');
```

The first argument shows how to use the filename to which to write and the Boolean overwrite setting. The filename must be given with a full path. The second argument is optional.

Clearing the Target Configuration

The following example shows how to use the `config_clear` function, which takes no arguments:

```
$error = $session->config_clear;
```

Getting a List of Recent Configuration Commits

The following example shows how to use the `config_get_history()` function that takes the optional arguments *Maximum* and *RollbackOnly*, which can be specified in any order by using the standard Perl hash notation:

```
$response = $session->config_get_history(Maximum => 10, RollbackOnly => 'true');
```

It returns a response object in which the method `get_entries` can be called. This returns an array of entry objects from which the `get_key` method can be called to retrieve the `CommitID` and `get_data` to retrieve the rest of the fields.

Rolling Back to a Previous Configuration

The following example shows how to use the `config_rollback()` function that takes the optional arguments *Label* and *Comment*, and exactly one of the two arguments *CommitID* or *Previous*:

```
$error = $session->config_rollback(Label => 'Rollback test', CommitID => 1000000072);
```

These arguments can be specified in any order using the standard Perl hash notation.

Getting a List of Current Configuration Sessions

The following example shows how to use the `config_get_sessions` function, which takes no arguments:

```
$response = $session->config_get_sessions;
```

It returns a response object in which the method `get_entries` can be called. This returns an array of entry objects in which the method `get_key` method can be called to retrieve the session ID and `get_data` method to retrieve the rest of the fields.

Sending a Command-Line Interface Configuration Command

The following example shows how to use the `config_cli()` function, which takes a string argument containing the CLI format configuration that you want to apply to the router:

```
$response = $session->config_cli($cli_command);
```

To retrieve the textual CLI response from the response object returned, you can use the `get_cli_response()` method, as shown in the following example:

```
$response_text = $response->get_cli_response();
```

**Note**

Apart from the `config_commit`, `config_get_history`, `config_get_sessions`, and `config_cli` methods, each of the other methods returns a reference to an error object if an error occurs or is undefined. For more information, see [“Using the Error Objects” section on page 15-135](#).

Using the Cisco IOS XR Perl Data Object Interface

Instead of having to specify the XML requests explicitly, the interface allows access to management data using a Perl notation. The Data Object interface is a Perl representation of the management data hierarchy stored on the router. It consists of objects of type `Cisco::IOS_XR::Data`, which corresponds to items in the IOS_XR management data hierarchy, and a set of methods for performing data operations on them.

To use the Data Object interface, a knowledge is needed of the underlying management data hierarchy. The management data on an IOS_XR router all under one of six `root` objects, namely `Configuration`, `Operational`, `Action`, `AdminConfiguration`, `AdminOperational`, and `AdminAction`. The objects that lie below these objects in the hierarchy, along with definitions of any datatypes or filters that are used by them, are documented in the Perl Data Object Documentation.

A hash structure is defined to be a scalar (that is, basic) type, for example, string or number, a reference to a hash whose values are hash structures, or a reference to an array whose values are hash structures. This standard Perl data structure corresponds naturally to the structure of management data on an IOS_XR router. The following example shows how to use a hash structure:

```
# basic type
my $struct1 = 'john';
# reference to a hash of basic types
my $struct2 = {Forename => $struct1, Surname => 'smith'};
# reference to an array of basic types
my $struct3 = ('dog', 'budgie', 'cat');
# reference to a hash of references and basic types
my $struct4 = {Name => $struct2, Age => '30', Pets => $struct3};
```

The following sections describe how to use the Perl Data Object Documentation:

- [Understanding the Perl Data Object Documentation, page 15-139](#)
- [Creating Data Objects, page 15-139](#)
- [Specifying the Schema Version to Use When Creating a Data Object, page 15-141](#)
- [Using the Data Operation Methods on a Data Object, page 15-141](#)
- [Using the Batching API, page 15-144](#)
- [Displaying Data and Keys Returned by the Data Operation Methods, page 15-145](#)
- [Specifying the Session to Use for the Data Operation Methods, page 15-146](#)

Understanding the Perl Data Object Documentation

The Perl Data Object Documentation consists of many files, and each contains a subtree of the total management data hierarchy. The main part of each filename tells you the area of management data to which that file refers, and the suffix usually tells you which root object that file's data lies below. For example, a file containing configuration data usually not end in `_cfg.html`. Some files may not contain any object definitions but just some data types or filter definitions, and usually do not end in `_common.html`.

For leaf objects, the object definition describes the data that the object contains. For nonleaf objects, the definition provides a list of the object's children within the tree. More precisely, the object definition consists of the following items:

- Name of the object.
- Brief description of what data is contained in the object or in the subtree below.
- List of the required task IDs that are required to access the data in the object and subtree.
- List of parent objects and the files in which they are defined, if the object is the top-level object in that file.
- If the object is a leaf object (for example, data is contained without child objects), and its name is not unique within that file, parent objects are listed.
- If the object is a table entry, a list of the keys that are needed to identify a particular item in that table. For each key, a name, description, and datatype are given.
- If the object is a table, a list of the filters that can be applied to that table.
- If the object is a leaf object, a list of the value items that are contained. For each value item, a name, description, and datatype are given.
- If the object is a leaf object, its default value (for example, the values for each of its value items that would be assumed if the object did not exist), if there is one.
- List of the data operation methods, `get_data`, `set_data`, and so forth that are applicable to the object. For more information, see [“Specifying the Schema Version to Use When Creating a Data Object” section on page 15-141](#)

Creating Data Objects

Data objects form a tree corresponding to a section of the data hierarchy. The first object to be created is one of the root data objects, and is created by a call to `Cisco::IOS_XR::Data::<object_name>`. For example, `<object_name>` is one of the following objects:

- Configuration
- Operational
- Action
- AdminOperational
- AdminAction

The following example shows how to create the Operational object:

```
my $oper = Cisco::IOS_XR::Data::Operational;
```

Because the syntax is rather lengthy for a task that is relatively common, there is a shorter way of creating a data object, which eliminates the need for the `Cisco::IOS_XR::Data::` at the front of the function name. This is achieved by importing the symbols for the root data object functions when using the `Cisco::IOS_XR` package at the top of the script. The following example shows how to import the Configuration and Operational functions:

```
use Cisco::IOS_XR qw(Configuration Operational);
```

The following example shows how to import all the root data objects without listing them explicitly:

```
use Cisco::IOS_XR qw(:root_objects);
```



Note

If there is a function in the script's name space with a name that is one of Configuration, Operational, and so forth, the root data objects cannot be imported with `use Cisco::IOS_XR qw(Configuration Operational)` and refer to the objects simply as Configuration, as this may not have the desired effect due to the ambiguity. Instead, you have to refer to them with the more lengthy `Cisco::IOS_XR::Data::Configuration` (that is, fully qualified) syntax.

If the root data object is Configuration, additional arguments can be specified that are given as name and value pairs. The *Source* argument can have values such as *ChangedConfig*, *CurrentConfig*, *MergedConfig* (the default value if the *Source* argument is not specified), and *CommitChanges*. If *CommitChanges* is specified, one of the two arguments *ForCommitID* and *SinceCommitID* must also be specified, as shown in the following example:

```
my $config = Configuration(Source => 'CommitChanges', ForCommitID => 1000083);
```

Data objects can be created from existing ones by calling a method on the existing object for which the name is that of the new object that you want to create. The object from which the new object was created is known as its parent, as shown in the following example:

```
my $config = Configuration;
my $bgp = $config->BGP;
```

If references to the intermediate objects are not required, the syntax allows a very compact way of creating objects as the methods can be strung together. The following example shows how to create a BGP object whose parent is Configuration:

```
my $bgp = Configuration->BGP;
```

If an object is an item in a table, its keys can be specified as arguments when the object is created by using the standard Perl hash notation. The following example shows how to create an object corresponding to the interface configuration for interface Ethernet 0/0/0/0:

```
my $if_conf = Configuration->InterfaceConfigurationTable->
    InterfaceConfiguration('Active' => 'act', 'Name' => 'Ethernet0/0/0/0');
```

Keys can also be specified by passing a hash structure as an argument. The hash structure would usually have been returned as a key from one of the data operation methods, for example, `get_keys`, but can be defined explicitly, as in the following alternative to the previous example:

```
my $key = {'Active' => 'act', 'Name' => 'Ethernet0/0/0/0'};
my $if_conf = Configuration->InterfaceConfigurationTable->InterfaceConfiguration($key);
```

There may be some occasions when it is better to keep references to the intermediate data objects, such as when you want to refer to more than one item in a table. The following example shows how to refer to more than one interface in the interface configuration table:

```
my $if_l_key = {'Active' => 'act', 'Name' => 'Ethernet0/0/0/0'};
```

```
my $if_2_key = {'Active' => 'act', 'Name' => 'POS0/4/0/0'};
my $if_conf_table = Configuration->InterfaceConfigurationTable;
my $if_conf_1 = $if_conf_table->InterfaceConfiguration($if_1_key);
my $if_conf_2 = $if_conf_table->InterfaceConfiguration($if_2_key);
```

Currently, there is no checking within the library that the object names specified are valid. However, when a data operation is performed on a data object, and if the object hierarchy is invalid, the response from the router should contain an error to this effect. For information on the valid object names in the data hierarchy, see [“Understanding the Perl Data Object Documentation” section on page 15-139](#).

Specifying the Schema Version to Use When Creating a Data Object

If you want to specify which version of a particular schema you are using, you may pass this information as arguments when creating the relevant data object. The router checks this information against its own schema versions when it receives a request, and rejects the request if the versions are not compatible. For more information about versioning, see [Chapter 9, “Cisco XML Schema Versioning.”](#)

The object in which this information should be specified is the top-level object within the schema whose version you want to specify. This information is found at the top of the page of the schema. For more information, see [“Understanding the Perl Data Object Documentation” section on page 15-139](#). For example, you may want to specify that using BGP schema version 1.4. The following example shows how to create a BGP object:

```
my $bgp = Configuration->BGP(MajorVersion => 1, MinorVersion => 4);
```

The object can then be used in the normal way to create child objects. Whenever any data operation request is sent using one of these objects, the specified version information is always included.

Using the Data Operation Methods on a Data Object

To access the management data on the router, data operation methods, which can be called on data objects, are provided for the getting, setting, and deletion of the corresponding data. The management session in which they act is the current session, and usually the most recent Cisco::IOS_XR object to be created. For more information on how to manually set which session to use for the data operation methods, see [“Specifying the Session to Use for the Data Operation Methods” section on page 15-146](#).

The types of data operation methods that are allowed depend on what the root data object is for the data object in question. For example, if the root object is Configuration, getting, setting, and deletion are allowed. If it is Operational, only getting is allowed. The get methods that can be used also depend on whether the data object in question is a leaf object or a table object.

Each of the data operation methods returns a response object from which any errors can be extracted. For more information, see [“Using Response Objects” section on page 15-134](#). For the methods that return values of some sort, a method of the same name is used to actually extract the information required from the response object.

get_data Method

The `get_data` method can be called on a leaf object and is used to retrieve the data contained in that object. It returns a response object from which the desired data can be extracted by calling the method of the same name, `get_data`.

The following example shows how to get the data for the interface configuration:

```
my $response = $if_conf->get_data;
if (defined($response->get_error)) {
    die $response->get_error;
} else {
    my $data = $response->get_data;
    ...
}
```

find_data Method

The `find_data` method performs a get request on a leaf object, but with the option of specifying key values for any table entries that occur within the hierarchy as a wildcard rather than as explicit values.

The XML response then contains every occurrence of the required object that matches the combination of key values and wildcards specified in the hierarchy.



Note

Wildcards are supported for only configuration data.

Currently, the function does not interpret the XML response in any way, due to the potentially complex structure of the returned data, and so the returned response object can be used only to extract the XML and any other errors in the usual way.

When specifying the keys for a table entry object, if you want one of the keys to be a wildcard rather than specified explicitly, pass an argument called `wildcard` value, where the value is the name of the key. If access control lists (ACLs) have been configured, the following example shows how to get the inbound ACLs of all interfaces on the router:

```
my $response = $if_conf_table->
    InterfaceConfiguration(Active => 'act', wildcard => 'Name')->
    IPV4PacketFilter->Inbound->find_data;
```

If you want one or more of the keys for a particular table entry to be wildcards, the value of the wildcard can be a reference to an array containing the names of those keys. For example, if you want to include any nonactive interface configuration in the above example, you would do the following:

```
my $response = $if_conf_table->
    InterfaceConfiguration(wildcard => ['Name', 'Active'])->
    IPV4PacketFilter->Inbound->find_data;
```

get_keys Method

The `get_keys` method must be called only on a table object and is used to retrieve a list of the keys for each item in the table. It returns a response object from which the keys can be extracted by calling the method of the same name, `get_keys`. This returns an array of hash structures containing the key values. A returned key can also be used as the parameter to a new data object.

The following example shows how to get the keys for each item in the configuration table and then for each key to create a data object and perform some operations with it:

```
my $response = $if_conf_table->get_keys;
if (defined($response->get_error)) {
    die $response->get_error;
} else {
    foreach my $key ($response->get_keys) {
        my $interface = $if_conf_table->InterfaceConfiguration($key);
        # do something with this object such as get_data...
    }
}
```

The following two optional arguments can be specified as name and value pairs:

- **Count**—Determines the maximum number of table entries that will be returned.
- **Filter**—Specifies a reference to a hash whose elements are the arguments to the filter plus an element `Filtername` that specifies the filter to use, as shown in the following example:

```
my $table = Operational->BGP->NeighborTable;
my $filter = {FilterName => 'BGP_ASFilter', AS => 6};
my $response = $table->get_keys(Count => 10, Filter => $filter);
```

get_entries Method

Similarly, the `get_entries` method must be called only on a table object, and is used to retrieve a list of the keys and data for each entry in the table.

It returns a response object from which the entries can be extracted by calling the method of the same name, `get_entries`. This method returns an array of entry objects. The `get_key` and `get_data` methods can then be called on an entry object to extract the key and data for that entry.

The following example shows how to get an array of the keys and data for each item in the interface configuration table and perform some operations with each:

```
my $response = $if_conf_table->get_entries;
if (defined($response->get_error)) {
    die $response->get_error;
} else {
    foreach my $entry ($response->get_entries) {
        my $key = $entry->get_key;
        my $data = $entry->get_data;
        # do something with these values...
    }
}
```

The same optional arguments, `Count` and `Filter`, can be specified in the `get_keys` method.

set_data Method

The `set_data` method is called only on leaf objects, and sets the data for the object in the specified argument. The argument must be a hash structure; for example, the data is returned by a previous call of `get_data` or `get_entries`.

The returned value is a response object from which the entries are extracted. Unless batching is enabled, the returned value is undefined.

The following example shows how to add a IPv4Multicast object to the GlobalAFTable of BGP AS 1 object:

```
my $data = {'Enabled' => 'true'};
my $global_af = Configuration->BGP->AS('AS' => 1)->Global->
    GlobalAFTable->GlobalAF('AF' => 'IPv4Multicast');
my $error = $global_af->set_data($data);
```

**Note**

- If not all items in a leaf object are specified when setting data, the remaining items are set to null (overwrites any value that may have been there previously).
- If the data is passed to set_data as a hash or basic type (not an array), it can also be provided explicitly rather than by reference, in the same way as keys can be specified.

The following example shows how data that is passed to set_data as a hash or basic type:

```
my $error = $global_af->set_data('Enabled' => 'true');
```

If the data to be set is an array, it must be provided by references because if it were given explicitly it would be incorrectly interpreted as a hash.

delete_data Method

The delete_data method can be used on any object, and deletes all data below that object in the hierarchy, as shown in the following example:

```
my $error = Configuration->BGP->AS('AS' => 1)->Global->delete_data;
```

The returned value is a response object from which any errors can be extracted. Unless batching is enabled, the returned value is undefined.

Using the Batching API

By default, whenever the set_data or delete_data methods are called on a data object, the resulting XML request is sent immediately. The script is enabled to verify immediately whether or not the operation was successful. However, if a script wants to set or delete many items at once, this can be a very inefficient method.

By using the batching API, a script can specify that it wants a group of set or delete operations all to be sent together in one XML request. This reduces the overhead of the router having to process multiple requests and reduces the amount of data that needs to be sent. Due to the way two XML requests with overlapping hierarchies are merged, the resulting XML is not as long as the sum of the original two. The common hierarchy is not repeated.

**Note**

A commit operation cannot be performed within a batch. To enforce this, the config_commit() function dies with an error if it is called while batching is in progress.

batch_start Method

When the `batch_start` method is called on the session object in question, all subsequent calls of `set_data` or `delete_data` are not performed immediately but are stored locally until the `batch_send` method is called. The following example shows how to enable batching on the session `$session`:

```
$session->batch_start;
```

**Note**

Any calls to `set_data` or `delete_data` between the `batch_start` and the subsequent `batch_send` methods return undefined rather than as a response object

batch_send Method

The `batch_send` method should be called at the point in the script when you want to send all set and delete operations that were made since the previous call to `batch_start`. The `batch_send` method sends these operations as a single XML request and returns a single response object. If this response contains no errors, all operations were successful. Otherwise, the details of any error returned must be analyzed to determine which operation caused the error, as shown in the following example:

```
my $response = $session->batch_send;
my $error = $response->get_error;
if ($error) {
    die "Error in batch_send: $error";
}
```

**Note**

An error occurs in the script if `batch_send` is called while batching is not in progress, for example, it must occur after a call to `batch_start`.

Displaying Data and Keys Returned by the Data Operation Methods

When a key or data is returned either by calling `get_data` or `get_keys` functions on a response object, or by calling `get_data` or `get_key` functions on an entry object that was returned from the `get_entries` function, it is always in the form of a value object. This object behaves identically to a hash structure; therefore, the value object can be easily navigated using hash and array dereferencing if required. A key value can be used when creating a new data object or as an argument to the `set_data` function.

However, if you want to display the whole structure or any parts of it, use the built-in function `to_string` on any value object that returns a formatted string form of the structure. In fact, you do not need to call the function `to_string` on the object. Using the value object in a scalar context, automatically converts it to a formatted string. The following code is shown:

```
my $response = Configuration->InterfaceConfigurationTable
->InterfaceConfiguration(Active => 'act', Name => 'POS0/2/0/0')
->get_data;
print $response->get_data;
```

The following example displays that data on the screen in a readable way:

```
Shutdown
  true
IPV4PacketFilter
  Inbound
    HardwareCount
    Name
      myacl
Description
  my POS interface
```

Specifying the Session to Use for the Data Operation Methods

If only one `Cisco::IOS_XR` object has been created, this management session is automatically used by subsequent data operation methods. In scripts in which more than one `Cisco::IOS_XR` object has been created, the data operation methods use whichever session is the current session. The session to use for the data operation methods is whichever `Cisco::IOS_XR` object was the last to be created, unless, you have since asked to change the current session by calling the method `use_for_data_operations` on the `Cisco::IOS_XR` object that you want to use.

The following example shows how to create two management sessions and then use the first one for subsequent data operations:

```
my $session1 = new Cisco::IOS_XR(host => 'router1');
# Here the current session is $session1
my $session2 = new Cisco::IOS_XR(host => 'router2');
# Here the current session is $session2
$session1->use_for_data_operations;
# Now the current session is $session1 again
```

Cisco IOS XR Perl Notification and Alarm API

The notification API provides functionality that enables a Perl script to register for and receive asynchronous responses or notifications during a management session on the router. One important type of notification is Alarms for which the specific API is provided.

The API allows a script to register, deregister, and receive alarms using Perl methods and objects. This completely hides the underlying XML from the user in much the same way that the data object API for normal management requests does.

The following sections describe how to use the Alarm API:

- [Registering for Alarms, page 15-147](#)
- [Deregistering an Existing Alarm Registration, page 15-147](#)
- [Deregistering All Registration on a Particular Session, page 15-147](#)
- [Receiving an Alarm on a Management Session, page 15-147](#)

Registering for Alarms

To register for a receipt of alarms on a particular management session, use the `alarm_register` function of the `Cisco::IOS_XR` object that represents the management session. The `alarm_register` function takes as arguments a list of name and value pairs, which specify the set of filter criteria that you want to use to filter the alarms that you receive. If no filter criteria are specified, all alarms are received. For a list of the valid filter criteria, see the “Alarm Registration” section on page 10-105 of Chapter 10, “Alarms.”

The following example shows how to register for receipt of all alarms of Group `SYS` and Code `CONFIG_I`:

```
my $response = $session->alarm_register(Group => 'SYS', Code => 'CONFIG_I');
```

The `alarm_register` function returns a response object that is checked for errors in a normal way. These errors may be returned if a value specified for one of the filter criteria is invalid.

In addition, a successful registration response contains a registration ID, which must be used if the script wants to deregister. In other words, cancel this registration. The registration ID can be extracted from the response object by calling the `get_registration_id` method, as shown in the following example:

```
my $registration_id = $response->get_registration_id;
```

Deregistering an Existing Alarm Registration

To deregister a particular registration, use the `alarm_deregister` function on the `Cisco::IOS_XR` object, by giving as an argument for the registration ID that was returned from the initial registration as follows:

```
my $response = $session->alarm_deregister($registration_id);
```

The response object that is returned is checked for errors to determine if the deregistration was successful.

Deregistering All Registration on a Particular Session

To deregister all alarm registrations that have been made on a particular management session, use the `alarm_deregister_all` function as follows:

```
my $response = $session->alarm_deregister_all;
```

The response object can be used to check for any errors. Currently, no errors should exist even if there was no registration to deregister on that session.

Receiving an Alarm on a Management Session

After alarms have been registered, the `alarm_receive` function can be called on the management session object. The `alarm_receive` function attempts to pick up an alarm from the transport, which may happen immediately if there is already an alarm waiting in the buffer. Otherwise, it waits until one is received. An optional timeout value can be specified as the argument. If an alarm is not received within the timeout limit, the function returns undefined. If no timeout value is specified, the default of an infinite timeout is used, as shown in the following example:

```
my $alarm = $session->alarm_receive(60); # Wait 60 seconds for an alarm
```

If an alarm is received within the timeout limit, the function returns an alarm object from which the following values can be extracted:

- RegistrationID—Specifies the registration ID that was returned from the registration for the matched alarm.
- SourceID
- EventID
- Timestamp
- Category
- Group
- Code
- Severity
- State
- CorrelationID
- AdditionalText

These values can be extracted using the corresponding `get_*` functions, as shown in the following example:

```
my $registration_id = $alarm->get_registration_id;
my $event_id = $alarm->get_event_id;
my $text = $alarm->get_additional_text;
```

Using the Debug and Logging Facilities

The following sections describe how to control debug and logging facilities within your script:

- [Debug Facility Overview, page 15-148](#)
- [Logging Facility Overview, page 15-149](#)

For more information on how to control debug and logging from the command line when starting a script, see [“Starting a Management Session on a Router” section on page 15-132](#).

Debug Facility Overview

The debug facility displays on the screen run-time information to aid investigation of problems. The user is given fine control over which debug messages are displayed to the screen by allowing the user to specify at any point in the script which types of debug they want to be displayed and which ones they do not.



Note

Debug applies to the script as a whole rather than to each management session.

Table 15-6 lists the current built-in types.

Table 15-6 Definitions for the Debug Types

Type	Description
transport	Specifies the messages relating to the state of the current transport, for example, Telnet or SSH.
xml	Displays the request and response XML for every request sent to the router that includes those generated by the Data Object interface and configuration services methods.
xml_response_parts	Displays each part separately if an XML response has been split into multiple parts.
user	Specifies that the script writer can be used to add his or her own debug messages.

To turn on debug, use the `Cisco::IOS_XR::debug_on` function at any point in your script, giving those types of debug that you want to turn on as arguments. This is shown in the following example:

```
Cisco::IOS_XR::debug_on('transport', 'xml');
```

Similarly, to turn off debug for certain types, use the `Cisco::IOS_XR::debug_off` function. Specifying no arguments turns off all types of debug, as shown in the following example:

```
Cisco::IOS_XR::debug_off('xml');
```

To insert your own debug messages in a script, use the `Cisco::IOS_XR::debug` function, giving as arguments the type of debug followed by the message. This is shown in the following example:

```
Cisco::IOS_XR::debug('user', 'This is a user debug message');
```

In addition to being able to use the built-in type `user` to add debug messages to the scripts, it is possible to define your own debug types to give greater control over what is displayed. This is done by calling the `Cisco::IOS_XR::add_debug_types` function and giving as arguments a list of name and value pairs. The name is the name of the new type, and the value is its display name (that is, the string that appears at the beginning of every message of that type when displayed on the screen at run time). This is shown in the following example:

```
Cisco::IOS_XR::add_debug_types('general' => 'General', 'detailed' => 'Detailed');
```

These types can immediately be used to write debug messages, as shown in the following example:

```
Cisco::IOS_XR::debug('detailed', 'This is a detailed debug message');
```

Logging Facility Overview

The logging facility leaves an audit trail of usage or diagnoses problems after an error has occurred. The types of logging messages that are supported include all debug types, including any user-defined debug types.

To turn on logging, use the `Cisco::IOS_XR::logging_on` function at any point in your script, giving those types of messages that you want to turn on for logging as arguments. This is shown in the following example:

```
Cisco::IOS_XR::logging_on('transport', 'xml');
```

Similarly, to turn off logging for certain types, use the `Cisco::IOS_XR::logging_off` function. Logging can be turned off for all types of messages by giving no arguments, as shown in the following example:

```
Cisco::IOS_XR::logging_off('xml');
```

By default, the messages will be written to a file called `ios_xr_log.txt` in the same directory as the running script. You can specify which file to use with the function `Cisco::IOS_XR::set_log_file` that can be called at any point in your script. For example, you may want to specify a different log file before carrying out operations on a different management session, as shown in the following example:

```
Cisco::IOS_XR::set_log_file('router2_log.txt');
```

In addition to being able to log each of the standard message types, the Telnet module allows two types of extra logging at a lower level. These can be turned on for the duration of a management session by specifying one of the following arguments when calling `Cisco::IOS_XR::new`, as listed in [Table 15-7](#).

Table 15-7 Logging Arguments

Type	Description
<code>telnet_input_log</code>	Logs all data received from the router, which usually includes the echoes of everything that is sent.
<code>telnet_dump_log</code>	Logs all I/O ¹ through the Telnet connection in a dump format. The dump, however, is less readable than the input log.

1. I/O = input/output

The value of each argument specifies the file to which the log should be written.



Note

If both types of logging are specified, the filenames must be different and they both must be different from the name of the standard log file.

Examples of Using the Cisco IOS XR Perl XML API

The following sections provide examples of using the Cisco IOS XR Perl XML API to perform some of the common router management tasks:

- [Configuration Examples, page 15-151](#)
- [Operational Examples, page 15-158](#)

The examples demonstrate the advantages of using the XML and Perl XML API instead of the CLI and existing screen-scraping techniques.

They are also intended to show how simple it is to convert the most common configuration and operational tasks to scripts using the Perl XML API, as well as how easy it is to write scripts to perform tasks that are not possible using the existing methods.

Some of these tasks may be quite involved, so sample scripts have been provided within the toolkit, which can be customized to suit your needs. Other tasks may require very few lines of code.

Those examples in which scripts have been provided have a line at the top of the script, which specifies the perl executable to use to run it. By default, this line is `#!/usr/bin/perl -w`. If this is not the location of perl on your machine, you must change this line accordingly before being able to run the script.

You may also need to give yourself execute permission on the script if it is not already set using the following **chmod** command:

```
chmod +x <script name>.pl
```

You should be able to run the script using the following command from the directory in which it resides:

```
./<script name>.pl
```

Configuration Examples

Examples are provided for setting the configuration and getting the running configuration, which are two of the most common configuration tasks. Additional examples cover the standard router applications. One of these examples also demonstrates in detail how you would use the Data Object documentation to help write the necessary code to access a particular item of data.

Another example shows how to use the Cisco IOS XR Perl Notification API to perform actions whenever particular events occur, such as getting the current configuration changes whenever a commit occurs, or sending an e-mail to notify an administrator when an interface is down.



Note

- In all basic examples of setting a configuration, the final step of committing the configuration is omitted to avoid repetition.
- All the examples are written as though the script begins with a `use` statement, which imports all root data object functions, such as Configuration, Operational, and Action, as shown in the following example:

```
use Cisco::IOS_XR qw(:root_objects);
```

If your script cannot import the functions due to name clashes, you must fully qualify the function names with the `Cisco::IOS_XR::Data::` prefix.

Setting the IP Address of an Interface

Setting the IP address of an interface is normally performed by a sequence of two CLI commands, as shown in the following example:

```
interface MgmtEth0/0/CPU0/0
ip address 1.2.3.4 255.255.255.0
```

To carry out this example in a Perl script using the Perl Data Object API requires only one line of code, although in practice you would usually break it up into smaller lines for clarity and to be able to reuse parts of it. This is shown in the following example:

```
my $config = Configuration;
my $if_conf_table = $config->InterfaceConfigurationTable;
my $eth0 = $if_conf_table->
    InterfaceConfiguration(Active => 'act', Name => 'MgmtEth0/0/CPU0/0');
$eth0->IPV4Network->Addresses->Primary->
    set_data(IPAddress => '1.2.3.4', Mask => '255.255.255.0');
```

If the script is needed to access some other configuration, it would not need to repeat the first line but could use `$config`. Similarly, if it is needed to access some other configuration associated with `Ethernet0/0/0/0`, it would not need to repeat the first three lines but could just use the `$eth0` variable. The following example shows how to set the MTU of the `Ethernet0/0/0/0` to 1500 interface:

```
$eth0->IPV4Network->MTU->set_data(1500);
```

**Note**

The code as shown in given examples would probably be used in the middle of a large script, which performs a more complex job. If it is a common task, it could also be wrapped in a small function for that purpose.

For example, a function to set up the IP address of an interface could be made very easily by using the preceding code, and is then called, as shown in the following example:

```
set_int_ip_address('Ethernet0/0/0/0', '1.2.3.4', '255.255.255.0');
```

The code could be wrapped in a small script that enabled the task to be performed from the command line, as shown in the following example:

```
set_int_ip_address.pl -name Ethernet0/0/0/0 -ip 1.2.3.4 -mask 255.255.255.0
```

Configuring a Simple BGP Neighbor

The following example shows a correspondence to the CLI commands and subcommands for configuring a Border Gateway Protocol (BGP) neighbor:

```
RP/0/RP0/CPU0:router# configure
RP/0/RP0/CPU0:router (config)# router bgp 1
RP/0/RP0/CPU0:router (config-bgp)# neighbor 1.2.3.4
```

The equivalence of these two commands using the Data Object interface is shown in the following example:

```
my $bgp_entity = Configuration->BGP->AS(AS => 1)->BGPEntity;
my $neighbor = $bgp_entity->NeighborTable->
    Neighbor(IPAddress => {IPV4Address => '1.2.3.4'});
```

The following example shows how to set the remote autonomous system (AS) number for the neighbor:

```
$error = $neighbor->RemoteAS->set_data(44);
```

You may want to set the description for this neighbor, as shown in the following example:

```
$error = $neighbor->Description->set_data('The router next door');
```

Adding a List of Neighbors to a BGP Neighbor Group

This is a more complex example, which shows how a script can be used to expedite a common task. You may want perform this task using the CLI. You would have to enter a series of commands, as shown in the following example:

```
RP/0/RP0/CPU0:router# configure
RP/0/RP0/CPU0:router (config)# router bgp 1
RP/0/RP0/CPU0:router (config-bgp)# neighbor 1.2.3.4
RP/0/RP0/CPU0:router (config-bgp-nbr)# use neighbor-group user1
RP/0/RP0/CPU0:router (config-bgp-nbr)# exit
RP/0/RP0/CPU0:router (config-bgp)# neighbor 2.3.4.5
```

```
RP/0/RP0/CPU0:router(config-bgp-nbr)# use neighbor-group user1
RP/0/RP0/CPU0:router(config-bgp-nbr)# exit
RP/0/RP0/CPU0:router(config-bgp)# neighbor 3.4.5.6
RP/0/RP0/CPU0:router(config-bgp-nbr) use neighbor-group user1

etc...
```

The sample shows how to perform this task in a faster and more user-friendly way, as shown in the following example:

```
./add_neighbors_to_group.pl -host my_router -user john
Password:
Neighbor-group: user1
Neighbor ip addresses to add:
1.2.3.4
2.3.4.5
3.4.5.6
<cr>
```

The script can be found in the examples/bgp/add_neighbors_to_group.pl file within the toolkit installation directory.

Displaying the Members of Each BGP Neighbor Group

The example shows how a script using the Cisco IOS XR Perl scripting toolkit can retrieve and display information in ways that cannot be done using the CLI on the router. The script allows you to display the current members of each neighbor group and, which groups oppose how the information can be viewed using the CLI. In the same way, the previous example shows you how to add neighbors to a group rather than to add the group to each neighbor.

The script can be found in the examples/bgp/display_neighbor_group_members.pl file.

Setting Up ISIS on an Interface

The simplest integrated Intermediate System-to-Intermediate system (ISIS) configuration task is to set up ISIS on an interface. The following CLI commands example is set up as though the interface in question is already configured:

```
RP/0/RP0/CPU0:router# configure
RP/0/RP0/CPU0:router(config)# router isis 1
RP/0/RP0/CPU0:router(config-isis)# net 49.0000.0000.3.00
RP/0/RP0/CPU0:router(config-isis)# interface POS0/2/0/0
RP/0/RP0/CPU0:router(config-isis-if)# address-family ipv4
```

To use the Data Object interface, define the ISIS instance. This is shown in the following example:

```
my $instance = Configuration->ISIS->Instance(InstanceID => 1);
```

The following example shows how to set up a Network Entity Title (NET) for that instance:

```
$instance->NETTable->NET(NET => '49.0000.0000.3.00')->set_data('True');
```

The following example shows how to set up the interface:

```
my $if = $instance->InterfaceTable->Interface(Name => 'POS0/2/0/0');
$if->Running->set_data('True');
```

The following example shows how to set up the IPv4 address family on that interface:

```
$if->InterfaceAddressFamilyTable->
  InterfaceAddressFamily(AF => 'IPv4', SubAF => 'Unicast')->
  Running->set_data('True');
```

Finding the Circuit Type That is Currently Configured for an Interface for ISIS

The example shows how to use the Perl Data Object documentation to help you write code to access a particular piece of data.

You may know that ISIS is configured on a particular interface, for example, POS 0/2/0/0, but you may not know whether that interface was configured as only Level 1, only Level 2, or both. The data you are looking for is ISIS configuration data, which should be documented in a file whose name ends with `_cfg.html`. A quick browse through the Data Object documentation files reveals `isis_cfg.html` as a sensible place to look.

The first object definition is ISIS. The parent object is specified as `RootCfg`, which is the top-level configuration object that is accessed using the `Configuration` function. The following example shows how to create an object that corresponds to ISIS configuration:

```
my $isis = Configuration->ISIS;
```

You found that the only child object of ISIS is `Instance`. Under the `Keys` heading, you notice that `Instance` has only one key called `InstanceID`. If the ISIS instance that you are interested in is 1, you can now create a data object corresponding to that instance by specifying the instance ID as an argument. This is shown in the following example:

```
my $instance = $isis->Instance(InstanceID => '1');
```

Browsing at the child objects of `Instance`, you see an object called `InterfaceTable`, and you want this item of data for a particular interface. Therefore, it is presumably somewhere under that object, as shown in the following example:

```
my $interface_table = $instance->InterfaceTable;
```

By looking at the definition of the `InterfaceTable` object, you see it has one child called `Interface`. Looking at the definition of `Interface`, you see that it has one child called `CircuitType`, which must be the item that you are looking for. The definition of `Interface` contains one key called `InterfaceName`, which specifies the interface to create the corresponding data object, as shown in the following example:

```
my $interface = $interface_table->Interface(InterfaceName => 'POS0/2/0/0');
```

The following example shows how to create a `CircuitType` object:

```
my $circuit_type = $interface->CircuitType;
```

Looking at the definition of `CircuitType`, you see that it has a value and the `get_data` method can be called on it. You can now retrieve the required data, as shown in the following example:

```
my $response = $circuit_type->get_data;
```

The actual value can now be accessed from the response object, as shown in the following example:

```
my $value = $response->get_data;
```

You may not want to perform so many steps, and it is probably not necessary. In practice, you may do some of the steps at once. The following sample code would do the same thing:

```
my $response = Configuration->ISIS->Instance(InstanceID => '1')
    ->InterfaceTable->Interface(InterfaceName => 'POS0/2/0/0')
    ->CircuitType->get_data;
my $value = $response->get_data;
```

Finally, you would have to know the type of value of CircuitType to do any comparison of the value, which is given as ISISConfigurableLevels. The definition of ISISConfigurableLevels states what values are valid for this item. The following enumerations are included with the valid values:

- Level1
- Level2
- Levels1And2

Configuring a New Instance, Area, and Interface for OSPF

The example shows how to set up OSPF on an interface.

Assuming that the POS0/2/0/0 and Loopback0 interfaces are already configured, the following example shows how to use the CLI commands:

```
RP/0/RP0/CPU0:router# configure
RP/0/RP0/CPU0:router(config)# router ospf 1
RP/0/RP0/CPU0:router(config-ospf)# router-id Loopback0
RP/0/RP0/CPU0:router(config-ospf)# area 1
RP/0/RP0/CPU0:router(config-ospf-ar)# interface POS0/2/0/0
```

By using the Data Object, you can define the OSPF process and instance that you are interested in and ensure that it is started. This is shown in the following example:

```
my $ospf_process = Configuration->OSPF->ProcessTable->Process(InstanceName => '1');
$ospf_process->Start->set_data('true');
```

The following example shows how to set up the router ID for the process:

```
$ospf_process->RouterID->InterfaceID->set_data("Loopback0");
```

The following example shows how to set up the area that this interface will be a part of:

```
my $area = $ospf_process->AreaTable->
    Area(IPAddressID => '0.0.0.0', IntegerID => 1, AreaIDFormat => 'integer');
$area->Running->set_data('true');
```

The following example shows how to configure the interface:

```
$area->NameScopeTable->NameScope(Interface => "POS0/2/0/0")->Running->
    set_data('true');
```

Getting a List of the Usernames That are Configured on the Router

You may want to get a list of the usernames that are configured on the router, but without all other information that is displayed by the CLI command **show aaa userdb**. The following example shows where you would use the `get_keys` function:

```
my $response = Configuration->AAA->UsernameTable->get_keys;
my @keys = $response->get_keys;
```

You could use the resulting array however you want; for example, to display your own compact list of usernames. This is shown in the following example:

```
print "Usernames configured on the system:\n";
foreach my $key (@keys) {
    print "$key->{Name}\n";
}
```

Finding the IP Address of All Interfaces That Have IP Configured

The example shows how to use the `find_data` function of the Data Object interface to find every occurrence of a particular leaf object that matches the combination of key values and wildcards that are specified in the hierarchy.

The code that is needed is almost identical to that which would be used to get the IP address of a particular interface, except that the `Name` key to the interface configuration table is specified as a wildcard, and the function calls the `find_data` method rather than `get_data` method. This is shown in the following example:

```
my $if_conf_table = Configuration->InterfaceConfigurationTable;
my $response = $if_conf_table->
    InterfaceConfiguration(Active => 'act', wildcard => 'Name')->
    IPV4Network->Addresses->Primary->find_data;
```

The XML response can be extracted from the returned response object using the `to_string` method. In addition, the XML response can be examined programmatically by extracting the DOM tree representation from the response object using the `get_dom_tree` method.

Adding an Entry to the Access Control List

The following commands show how to add an entry to an access control list (ACL) to block a particular source IP address:

```
RP/0/RP0/CPU0:router# configure
RP/0/RP0/CPU0:router(config)# ipv4 access-list user1
RP/0/RP0/CPU0:router(config-ipv4-acl)# deny ip host 1.2.3.4 any
```

The following commands show how to add an ACL to the inbound traffic of an interface:

```
RP/0/RP0/CPU0:router# configure
RP/0/RP0/CPU0:router(config)# interface POS0/2/0/0
RP/0/RP0/CPU0:router(config-if)# ipv4 access-group user1 in
```

You can also perform the tasks using the Perl Data Object API. The code acts as though the specified ACL already exists and the last sequence number in the list is already known. The last sequence number for the new entry can be easily calculated. The following example shows how to define the relevant tables that you are interested in:

```
my $acl_table = Configuration->IPV4_ACLAndPrefixList->AccessListTable;
my $if_conf_table = Configuration->InterfaceConfigurationTable;
```

The following example shows how to add a new entry to the ACL. (This request sets all other items in an access list entry to null.)

```
my $acl = $acl_table->AccessList(Name => 'user1');
$error = $acl->AccessListEntryTable->
    AccessListEntry(SequenceNumber => '50')->
    set_data(SourceAddress => '1.2.3.4',
            Grant => 'Deny',
            Protocol => 'IP');
```

The following example shows how to add the ACL to the interface:

```
my $interface = $if_conf_table->
    InterfaceConfiguration(Active => 'act', Name => 'POS0/2/0/0');
$error = $interface->IPV4PacketFilter->Inbound->set_data('user1');
```

Denying Access to a Set of Interfaces from a Particular IP Address

The intended use of the script is to quickly and easily block a particular IP address from gaining access to the router on whichever interfaces that you choose, for example, a security threat. This is a good example of a script that retrieves some existing configuration data. Based on the information, some new configuration is applied to the router.

In practice, the set requests are all sent in one request as described in the following list:

- The interfaces in which the new ACL entry is to be applied, the IP address to block, and the name of the new ACL (if needed) are entered by you when prompted. If desired, `all` can be specified instead of listing all interfaces on the system.
- The router is queried to see which access control lists are defined, and what the current entries are to find the last sequence number in each list.
- The router is queried to see which inbound ACLs are assigned to each interface, if any.



Note The request retrieves only the specific configuration that is desired, which can be done by using the CLI. In addition, it makes use of a wildcard for the Name key of the interface table to get the required data for all interfaces, which can also be done by using the CLI.

- If any of those interfaces did not already have an ACL assigned to it, a new ACL is created and assigns it to those interfaces.
- The new ACL entry is added to each of the existing ACLs that were assigned to one of the interfaces in question, and to the new ACL if there is one.



Note The example could be easily extended to block more than one IP address, or to apply the new ACL entry to multiple routers at one time. The script can be found in the `examples/acl/deny_access.pl` file.

Each configuration item is set with an individual call to the `set_data` function of the Data Object interface. Usually, this would result in many separate XML requests. Because the batching API is used, the configuration is set using a single XML request to maximize efficiency.

Performing Actions Whenever Certain Events Occur for Notifications

The sample, which demonstrates how to use the Cisco IOS XR Perl Notification and Alarm API, shows how to perform an action whenever a certain event occurs. In particular, it informs someone through e-mail (network administrator) about the events listed in [Table 15-8](#).

Table 15-8 List of Events

Event	Description
Interfaces going up/down	When the event occurs, an e-mail is sent informing the recipient of the interface, which router the interface is on, and the new state.
Configuration change	When the running configuration on the router is changed, an e-mail is sent informing the recipient that the event occurred. The configuration change event includes the commitID of the latest commit, the location of a file that contains the commit changes in XML format, and a readable version of the commit changes.

The following steps for the script are described:

1. Registers for alarms for the two relevant types, which are determined by specifying the Group and Code fields, and records the two returned registration IDs.
2. Enters an event loop in which the script calls the alarm_receive function to get the next alarm from the session and calls the relevant handler determined by the registration ID of the alarm.

For change in configuration, differences are retrieved from the router using the same management session that is used for receiving alarms. The XML response is stored in a local file with each commit being stored in a separate file. A readable version of the differences, which is created automatically by using the data object in a string context, is included in the e-mail.

An e-mail, is sent to the specified address, which may be regular e-mail or a message sent to a pager. This is not practical for a long message (for example, a configuration change), but would be well-suited to a single-line message similar to the interface up/down case.

In essence, actions taken when an event occurs are not limited to sending e-mails. A script could do just about anything in response to an event; for example, performing actions or changing configuration on the router. In addition, a script could register to receive notifications from more than one router, which gives it the ability to know the state of a whole network and perform actions accordingly.

The script can be found in the examples/notification/notification.pl file.



Note

The script uses the Perl module Mail::Send, which must be installed to use it.

Operational Examples

Some examples can be used to retrieve operational data from the router. These examples all make use of the Perl Data Object API due to the ease with which requests can be formed, and the flexibility of having access to a Perl representation of the response data and the XML form.

Some examples are very simple, such as retrieving all data in a particular table, and requires only a couple lines of code. Other examples involve getting data from more than one place and combining the data somehow. An example script is provided.

There are some scripts that give examples of how to display the retrieved data in different ways. First are some examples of producing a textual output similar to the corresponding CLI command. These use the Data Object interface because of the ease with which the desired data can be extracted from the Perl representation of the response.

Some examples can be used to transform an XML response into an HTML table for easy viewing in a web browser. These examples take full advantage of having the response data in XML format. HTML can be produced very easily from XML using the style sheet transformation language XLST.

Retrieving the Operational Information for All Interfaces on the Router

The following example shows how to retrieve a single table of data, which can be done very easily by using the Data Object interface `get_entries` function. It can be done all in one line (or rather one statement that has to be split over multiple lines). For clarity and to take advantage of error checking however, it is best to do the retrieval in stages.

To retrieve the operational information for all interfaces on the router, perform the following steps:

-
- Step 1** Define the key for the data node that you are interested in (for example, primary RP), as shown in the following example:

```
my $data_node = {RPLocation => {Rack => 0, Slot => 0, Instance => 'CPU0'}};
```

- Step 2** Define the table and what contents to retrieve, as shown in the following example:

```
my $interface_table = Operational->InterfaceProperties->DataNodeTable->
    DataNode($data_node)->SystemView->InterfaceTable;
```

- Step 3** Call the `get_entries` function on the table, as shown in the following example:

```
my $response = $interface_table->get_entries;
```

- Step 4** Check to see if an error occurred. If not, retrieve the entries, as shown in the following example:

```
if (!defined($result->get_error)) {
    my @entries = $result->get_entries;
    # Now do something with the entries...
}
```

Retrieving the Link State Database for a Particular Level for ISIS

Another example is to retrieve a single table of data, which is accomplished by using the `get_entries` function. The data that is retrieved corresponds roughly to that displayed by the CLI command **show isis database level <level no>** and split up into three stages. You will notice that the last two stages are exactly the same.

To retrieve the link state database for a particular level for ISIS, perform the following steps:

-
- Step 1** Define the table from which you want to get data from, as shown in the following example:

```
my $table = Operational->ISIS->Instance(InstanceID => 1)->LevelTable->
    LevelInstance(Level => "Level1")->LSPTable;
```

- Step 2** Call the `get_entries` function on the table, as shown in the following example:

```
my $response = $table->get_entries;
```

Step 3 Check to see if an error occurred. If not, retrieve the entries, as shown in the following example:

```
if (!defined($result->get_error)) {
    my @entries = $result->get_entries;
    # Now do something with the entries...
}
```

Getting a List of All Interfaces on the System

The simple example shows how to use the `get_keys` function to get a list of the items in a table without getting all the other associated data with them, which cannot be done using the CLI **show** commands.

To get a list of all interfaces on the system, perform the following steps:

Step 1 Define the table, as shown in the following example:

```
my $interface_table = Operational->InterfaceProperties->DataNodeTable->
    DataNode(RPLocation => {Rack => 0, Slot => 0, Instance => "CPU0"})->
    SystemView->InterfaceTable;
```

Step 2 Call the `get_keys` function and check for errors, as shown in the following example:

```
my $response = $interface_table->get_keys;
if ($result->get_error) {
    die "Error in get_keys: $result->get_error";
}
```

The list of interfaces is returned now as an array from `$response->get_keys` to carry out an action for each interface.

The following example shows how to print it:

```
foreach my $if ($result->get_keys) {
    print $if->{Name} . "\n";
}
```

Retrieving the Combined Interface and IP Information for Each Interface

This is a more complicated example of retrieving operational data as it gets the data from more than one place and combines that data in some way. The `get_ip_interfaces()` function retrieves the operational state for each interface and the IPv4 information for each interface that has IPv4 information.

These two sets of information are combined into one table so that the data is easily accessed by a script that wants to display the data. This is exactly the information that is used by the CLI command **show ip interfaces**.

The function is a good example of how the use of XML makes scripts robust to changes in the underlying data. For example, if new data items are added to the tables, or names of items change, the function still works. The function can be found in the `examples/interfaces/get_ip_interfaces.pm` file within the toolkit installation directory.

Listing the Hostname and Interface for Each ISIS Neighbor

You can call the `get_keys` function on the ISIS neighbor table that returns the interface name and system ID for each neighbor. The system ID is an internal value that uniquely identifies the neighbor, but it is not very useful as a displayed value. However, a second table called the *hostname table* provides a mapping from the system ID to the actual hostname (amongst other things). The hostname is displayed by the CLI `show` command **show isis neighbors**, rather than by the system ID. Thus, combining the data from these tables, you can produce a list of the hostname and interface name for each neighbor.

The `list_isis_neighbors` function example resides in the `examples/isis/list_isis_neighbors.pm` file. The function calls the `get_keys` function on the `NeighborTable` object, which produces a list of the system ID and interface for each neighbor. Then, it calls the `get_entries` function on the `HostnameTable` object, and maps the resulting table into a hash, which provides a mapping from system ID to hostname. Finally, the mapping is used to create a new array containing a list of the hostname and interface for each neighbor.

As an example of using the `list_isis_neighbors` function, the following code prints the hostname and interface for each neighbor for ISIS instance 1:

```
require '<toolkit inst dir>/examples/isis/list_isis_neighbors.pm';
my @neighbors = list_isis_neighbors(1);
print "Interface:      Hostname:\n";
foreach my $nbr (@neighbors) {
    printf("%-20s%s\n", $nbr->{Interface}, $nbr->{Hostname});
}
```

Recreating the Output of the show ip interfaces CLI Command

The example shows how to write an easily customized script for displaying information retrieved from a router.

The script gets the required data by calling the `get_ip_interfaces()` function. For details, see [“Retrieving the Combined Interface and IP Information for Each Interface” section on page 15-160](#). The script goes through each entry in the table and picks particular data items, and displays them in a custom format that is the same format as the original **show** CLI command.

The display function easily can be customized by removing sections when data is no longer of interest, adding sections if new data needs to be displayed, or changing the way particular data is displayed. You can create your own version of the **show ip interfaces** CLI command.

The function is clearly more dependent on the names of the data items that are returned and their formats than the underlying `get_ip_interfaces()` function. Because of XML, the function still works if extra items are added to or removed from tables that are not currently being displayed.

The script can be found in the `/examples/interfaces/show_ip_interfaces.pl` file within the toolkit installation directory.

Producing a Textual Output Similar to the show bgp neighbors CLI Command

This is another example of displaying data retrieved from the router in a custom format and again in the same style as the original **show** CLI command. The data-retrieval part of the script is simple and uses the `get_entries` function, as shown in the following example:

```
my $response = Operational->BGP->NeighborTable->get_entries;
```

The script goes through each of the returned entries and calls the `print_neighbor_info` function to display the details of the neighbor.

The function shows how easily the required items can be accessed and displayed, and how to ignore information in which you are not interested (for example, AF-specific information, which uses the Data Object interface).

The script can be found in the examples/bgp/show_bgp_neighbors.pl file within the toolkit installation directory.

Displaying Tabular XML Data in a Generic HTML Table Using XSLT

HTML is one of the most useful ways to display data. The HTML format has many useful features, such as the ease in which it can display formatted data in a platform-independent way, and the ability to add links for easy navigation.

For data that takes the form of a list of records (for example, a table), an HTML table is a natural way to display it. A sample function has been provided that uses XSLT to transform a table of data in XML format into an HTML table.

The function is generic. You can pass as an argument the name of the table that you want to display, and the script automatically tries to produce the best HTML table it can. If the table is simple (for example, each field in the table has exactly one value), the output should be a good representation of the data.

If the structure of the data is more complicated (for example, certain fields contain multiple subfields or even subtables within the table), the contents of these subfields or subtable appears within one field and probably will not be very useful.

The function can be found in the examples/common/xml_to_html_table.pm file within the toolkit installation directory. The XSL file it uses to do the transformation is in examples/common/xml_to_html_table.xsl.



Note

The XML::LibXSLT module must be installed to use the example.

To use the function, you may want to display the operational data for each interface on the router as follows:

Step 1 Use the require statement to specify the name of the module, as shown in the following example:

```
require "xml_to_html_table.pm"; # may need to specify a path here
```

Step 2 Retrieve the information in XML format. The following example uses the Data Object interface to do this:

```
my $data_node_table = Operational->InterfaceProperties->DataNodeTable;
my $interface_table = $data_node_table->DataNode(RPLocation =>
    {Rack => 0, Slot => 0, Instance => "CPU0"})->SystemView->InterfaceTable;
my $response_string = $interface_table->find_data->to_string;
```

Step 3 Transform the XML to HTML, as shown in the following example:

```
xml_to_html_table($response_string, $html_file, "InterfaceTable");
```

The example can be found in the examples/interfaces/generic_interface_props_table.pl file.

The resulting HTML file contains a table with a row for each interface and a column for each field of data. Clearly, the function is best suited for tables in which the number of fields is small enough that they all fit the screen. However, the main drawback to using the generic function is that the display format of

the fields is identical to XML, which may not be desired—as is the case with the Type, State, and Line State fields in the interface properties example. For more information, see [“Displaying the Interface State in a Customized HTML Table” section on page 15-163](#).

Displaying the Interface State in a Customized HTML Table

In many cases, the generic HTML table example does not display the information quite as you want it. For example, you may want to display only some data items for each table entry and change the display format of certain items.

The example produces an HTML containing the State and Line State for each interface and ignores all other data in the interface table. The following enhancements are provided over the generic HTML table:

- Only the fields of interest are displayed, and they are displayed in the order desired rather than the order that they appear in XML.
- The State and Line State fields are converted from their numeric values to text values that are easier to understand.
- Color-coding is added so that important information, such as an interface being down, stands out.
- The interfaces are sorted, so any interfaces that are down appear before those that are up—which makes it easy to spot problems without having to scroll down a long list.

The transformation is again done using XSLT. The XML::LibXSLT module must be installed to run the example. This means that the Perl script is very short and most of the work is done in the XSL file, which can be easily modified to customize the format of the displayed data, or extended to display more of the information returned in the XML.

The request in the example is stored as preformed XML to demonstrate the use of the basic Perl XML API, but the script could easily have been written using the Data Object API to form the request.

The example can be found in the examples/interfaces/interface_props_table.pl file. The XSL style sheet can be found in the examples/interfaces/interface_props_table.xsl file.

Displaying the BGP Neighbor Operational Data in a Complex HTML Format

This is an example of displaying data that does not conform naturally to a simple table format. The data displayed corresponds roughly to the command-**show bgp neighbors**, for which the output has an entry for each BGP neighbor and within each entry a subtable of information exists for each address family. Because the intended use of the script is for monitoring, the only values shown are operational rather than configurational.

Unlike the previous example, the script uses the Perl Data Object API to create the request that avoids having to write to any XML, which makes it quicker to write and easier to understand and maintain. However, the response format used is still XML and is needed to transform into HTML using XSLT.

The layout of the HTML output has a structure similar to that of the **show** command, with each BGP neighbor entry consisting of a selection of items laid out in a logical way, which includes the address family information that is displayed in a simple subtable.

The benefits of having the information in HTML are as follows:

- The format of bold headings makes the information clearer.
- The layout is much easier to control using tables, because they automatically adjust themselves to fit the information contained in them and the available space on the screen.

To facilitate navigation of the neighbor list, a separate HTML page is created that contains a simple summary table, with one entry for each neighbor. Each neighbor in the table has a link, which when clicked jumps to the neighbor's entry in the main table.

When the script is run, a session is created on the router that repeatedly polls the router for the latest information at regular intervals by updating the HTML files each time. Each of the two HTML files causes the web browser to automatically refresh them at the same regular interval, so the values on screen are automatically kept up to date. Ideally, the script would be modified to run as a CGI script on a web server, so that you can just open the web page (from any machine that has access) and not have to start the script first.

The script can be found in the `examples/bgp/bgp_neighbor_table_html.pl` file. The summary page produced is `examples/bgp/bgp_neighbor_table_summary.html`, and the main page is `examples/bgp/bgp_neighbor_table.html`.

**Note**

The XML::LibXSLT module must be installed to run the example.
