# Oracle® Communications IP Service Activator

Configuration Development Kit Guide

Release 7.3

**E61102-02**

June 2016

ORACLE®

Oracle Communications IP Service Activator Configuration Development Kit Guide, Release 7.3

E61102-02

Copyright © 2011, 2016, Oracle and/or its affiliates. All rights reserved.

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, then the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, delivered to U.S. Government end users are "commercial computer software" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, use, duplication, disclosure, modification, and adaptation of the programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, shall be subject to license terms and license restrictions applicable to the programs. No other rights are granted to the U.S. Government.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information about content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services unless otherwise set forth in an applicable agreement between you and Oracle. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services, except as set forth in an applicable agreement between you and Oracle.

# Contents

**Preface** ..... vii

Audience ..... vii
Accessing Oracle Communications Documentation ..... vii
Documentation Accessibility ..... vii
Document Revision History ..... vii

**1 Configuration Development Kit Overview**

**About the Configuration Development Kit** ..... 1-1
**Restrictions** ..... 1-2
**Pre-Defined Scripts** ..... 1-2

**2 Driver Scripts**

**About Driver Scripts** ..... 2-1
**About Script Types and Targets** ..... 2-1
Applying Roles ..... 2-2
Policy Targets ..... 2-2
**About Context-Specific Parameters** ..... 2-2
Local Context ..... 2-2
Inherited Context ..... 2-3
**About Scheduling of Scripts** ..... 2-3
**About Running Scripts** ..... 2-4
Scripts to Install Configuration ..... 2-4
Scripts to Remove Configuration ..... 2-5
**About Sharing Data Between Scripts** ..... 2-6

**3 Using Existing Scripts**

**Importing Scripts** ..... 3-1
**Viewing and Organizing Scripts** ..... 3-2
Creating Driver Script Folders ..... 3-3
Viewing a Summary of Scripts ..... 3-3
Viewing the Entire Text of Scripts ..... 3-5
Viewing Script Properties ..... 3-5
**Associating Roles with Scripts** ..... 3-6
**Setting Variables** ..... 3-7
Setting Variables in the Preamble Section ..... 3-7

Setting Variables in the Local Context for an Object .......... 3-8
**Associating Scripts with Objects** .......... 3-9
Linking Scripts to Objects .......... 3-9
Re-running a Script .......... 3-9
**Removing Scripts** .......... 3-10
**Propagating Configuration** .......... 3-10
Applying Configuration .......... 3-11
Removing Configuration .......... 3-11
**Deleting Scripts** .......... 3-11

## 4 Developing Scripts

**About Developing Scripts** .......... 4-1
**Creating a Script** .......... 4-1
Using the Template File .......... 4-1
Creating a Script Using a Text Editor .......... 4-2
Creating a Script from the IP Service Activator Client .......... 4-2
**Structure of a Script** .......... 4-3
The Preamble Section .......... 4-4
The Behavior Section .......... 4-5
The Common Section .......... 4-6
The Install Section .......... 4-7
The Remove Section .......... 4-8
**Exporting Scripts** .......... 4-9
**Programming Tips** .......... 4-9
Script Conventions .......... 4-9
Command Format .......... 4-10
Handling Exceptions .......... 4-10
Displaying Errors in IP Service Activator .......... 4-10
Applying Commands .......... 4-10
Preventing Command Application .......... 4-11
Processing Command Output .......... 4-11
Returning a Result .......... 4-11
Re-applying Configuration .......... 4-12
Managing Script Context .......... 4-12

## 5 Sharing Data Between Scripts

**About Sharing Data Between Scripts** .......... 5-1
**Using Classes** .......... 5-1
**Using a Python Dictionary** .......... 5-3
**Storing and Retrieving Data** .......... 5-4
**An Example of Using the Shared Data Area** .......... 5-5
Types of Script .......... 5-5
About the Example .......... 5-6
Data Script: setIPAddress.py .......... 5-10
Listing of setIPAddress.py .......... 5-10
Explanation of setIPAddress.py .......... 5-11
Data Script: setDescription.py .......... 5-11

Listing of setDescription.py ..... 5-11
Behavior Script: processInterfaces.py ..... 5-12
Listing of processInterfaces.py ..... 5-12
Explanation of processInterfaces.py ..... 5-14
Controller Script: BehaviorAndCommandController.py ..... 5-16
Listing of BehaviorAndCommandController.py ..... 5-16
Explanation of BehaviorAndCommandController.py ..... 5-16
**Error Reporting in Behavior Scripts** ..... 5-17

## 6 Monitoring and Troubleshooting Scripts

**Checking the Status of Scripts** ..... 6-1
**Understanding Warnings and Error Messages** ..... 6-2
**Checking Logs** ..... 6-3

## 7 Definition of Standard Methods

**Summary of Methods** ..... 7-1
**General Context** ..... 7-2
The _result Object ..... 7-2
The setCode Method ..... 7-3
The setDetails Method ..... 7-3
The sendScriptObjectFailure Method ..... 7-4
**Device Context** ..... 7-4
The _device Object ..... 7-4
The openSession Method ..... 7-5
The deliverCommand Method ..... 7-5
The closeSession Method ..... 7-6
The getIpAddress Method ..... 7-6
The getIos Method ..... 7-6
The getOs Method ..... 7-7
The getDeviceType Method ..... 7-7
The getFeatureSet Method ..... 7-7
The getNumberOfInterfaces Method ..... 7-8
The getThisCommitSharedData Method ..... 7-8
The getLifetimeSharedData Method ..... 7-8
The getInterface Method ..... 7-9
The log Method ..... 7-9
The auditLog Method ..... 7-10
**Interface Context** ..... 7-10
The _interface Object ..... 7-10
The getInterfaceName Method ..... 7-10
The getIpAddress Method ..... 7-11
The getVipType Method ..... 7-11
The getAdapterType Method ..... 7-12
The getNumberOfFramePvcs Method ..... 7-12
The getNumberOfAtmPvcs Method ..... 7-12
The getFramePvc Method ..... 7-13

The getAtmPvc Method ........ 7-13
**ATM PVC Context** ........ 7-14
The _atm_pvc Object ........ 7-14
The getVpi Method ........ 7-14
The getVci Method ........ 7-14
**Frame PVC Context** ........ 7-14
The _frame_pvc Object ........ 7-15
The getDlci Method ........ 7-15

## A Pre-defined Scripts

**Save to NVRAM** ........ A-1
**Add VLAN to CatOS** ........ A-1
**Force a FastStart Mode Exit for Cisco Devices** ........ A-2

## B Sample Scripts for Using the Shared Data Area

**Sample Python Module** ........ B-1
**Sample Behavior Script** ........ B-3
**Sample Data Script** ........ B-4
**Sample Controller Script** ........ B-4

# Preface

This guide describes how to develop scripts to implement device-specific network configuration.

The CDK enables system developers to develop Python scripts that define specific network configuration. These scripts can then be applied directly to network devices by using Oracle Communications IP Service Activator.

## Audience

This guide is intended for system developers who want to develop scripts to implement device-specific network configuration, and for network operations managers who use the scripts.

Before reading this guide, you should have familiarity with IP Service Activator and its QoS, access control, and VPN provisioning features.

You should also have knowledge of the Python programming language, as this guide does not provide details of the language.

## Accessing Oracle Communications Documentation

IP Service Activator for Oracle Communications documentation, and additional Oracle documentation, is available from Oracle Help Center:

http://docs.oracle.com

## Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

### Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

## Document Revision History

The following table lists the revision history for this guide.

| Version | Date | Description |
| --- | --- | --- |
| E61102-02 | June 2016 | Documentation updates. |
| E61102-01 | July 2015 | Initial release. |

# 1
# Configuration Development Kit Overview

This chapter provides an overview of the Configuration Development Kit (CDK) and the concept of driver scripts.

## About the Configuration Development Kit

The CDK is an optional Oracle Communications IP Service Activator module that extends the capabilities of the core system. It provides a means of defining device configurations through command scripts that are applied directly to network devices. The CDK supports Python version 2.2, which is automatically installed with IP Service Activator.

The CDK provides a method of implementing configuration tasks that could otherwise not be achieved from within the core system. For example, scripts can be written to create interfaces or sub-interfaces, set up IP addresses or run device-specific routing commands.

Key features of the CDK are as follows:

- Scripts are written in Python v2.2, a freely-available and fully-featured object-oriented language that is easy to learn.
- No programming experience is required to set up and apply existing scripts. Users can view existing scripts directly from the IP Service Activator user interface (client), set up any variables required and associate the scripts with appropriate devices, interfaces or VPNs in the network.
- Experienced Python developers can produce their own scripts. New scripts can be entered directly into the user interface; alternatively scripts can be written using a text editor and imported.
- The CDK is supported by all the fully-featured device drivers and by the CatOS script driver.
- Each script has a defined type, which specifies whether it configures devices, interfaces, sub-interfaces, ATM PVCs or Frame Relay PVCs. However, a script can be applied to objects throughout the network topology (networks, VPNs, devices, interfaces, sub-interfaces, and PVCs) and is applied by a process of inheritance.
- User-defined variables can be associated with the particular objects to which scripts can be applied, so for example you can define a generic script that, when run, applies different values to different interfaces.
- Users can select when a script is run. For example, a script can be run either before or after any other configuration tasks, and can be run once only or repeated at every subsequent propagate.

- Data can be shared between the scripts that are run for a particular device, which has a range of applications including control of script application order and optimization of script usage.

## Restrictions

Users of the CDK should be aware of the following:

- Commands applied via a script are not processed by the policy server which means that they are not checked or validated in any way. Applying untested scripts, or inadvertently applying scripts to the wrong devices can have a serious effect on network functionality, for example by removing routing capability.
- The ability to create or apply driver scripts is dependent on a user's access level. For security reasons, Oracle advises that access should be restricted to a small set of trusted users.

## Pre-Defined Scripts

As of IP Service Activator 7.2, two pre-defined scripts are supplied:

- **NVRAM.py**: This Cisco driver script copies the running IOS configuration to startup configuration.
- **CatosAddVlan.py**: This CatOS driver script adds VLAN to a CatOS switch.

For full details, see "Pre-defined Scripts".

# 2

# Driver Scripts

This chapter provides a general explanation of driver scripts and how they can be used.

## About Driver Scripts

A driver script is a short program, written in the Python programming language, which is used to directly configure a device via the Oracle Communications IP Service Activator device drivers. Python allows the use of loops and if/then statements when generating device commands.

Scripts can be created directly from within the IP Service Activator user interface (client) and are stored in the system database. These scripts can then be associated with specific targets, such as a device or an interface. For each object that the script is applied to, the device driver executes the script and generates and applies the configuration. The timing of the configuration is dependent on control information, which is defined as part of the script.

A driver script is divided into five sections:

- Preamble section: contains variables required in the rest of the script.
- Behavior section: includes parameter definitions and scheduling information which define where and when a script is applied.
- Common section (optional): defines functions that can be used more than once in the script, such as finding an IP address or returning the device status.
- Install section: contains code that is run when a script is associated with an object in order to configure commands on a device.
- Remove section (optional): contains code that is run when a script is disassociated from an object in order to remove configuration from a device.

## About Script Types and Targets

There are five different types of script, each with a particular target:

- Device: applies commands at device level
- Interface: applies commands at interface level
- Sub-interface: applies commands at sub-interface level
- ATM PVC: applies commands to ATM PVCs
- Frame Relay PVC: applies commands to Frame Relay PVCs

### Applying Roles

In common with rules and PHB groups, driTaver scripts must be allocated roles in order to define the points in the network at which they will be applied.

All scripts must have one or more device roles. Interface, sub-interface and PVC scripts must also have one or more interface roles assigned.

A script is only implemented on objects that have the same role. For example, a Device script with the role of Core device is only applied on those devices that are assigned the role of Core. For a Sub-interface script, the device, interface and sub-interface roles must all match for the script to be applied. For information about using roles in policy elements, see *IP Service Activator QoS User's Guide*.

### Policy Targets

Scripts can be associated with objects at different levels of the hierarchy and are then applied to all relevant objects by a process of inheritance. For example, if a sub-interface script is associated with a network object, it is applied to all sub-interfaces on devices within that network that have the correct role setting.

When the device driver runs the script, a separate concrete script is created for each instance.

You can associate scripts with the following objects:

- Customers
- VPNs
- Sites
- Networks
- Devices
- Interfaces
- Sub-interfaces
- PVC endpoints

Figure 2–1 shows a diagram of device script inheritance.

> **Note:** In situations where more than one script applies to an object, no priority ordering is carried out. You should ensure that multiple scripts do not result in conflicts.

## About Context-Specific Parameters

Each script runs in a specific context. This ensures that the writer of a script can assume the existence of a number of certain defined objects, and set up particular parameters. For example, at the device level, all device-specific information, such as IP address, device type and number of interfaces, is automatically available. At the interface level, all interface-specific information is available in addition to device details.

### Local Context

Each object that can have a driver script applied to it can have additional commands defined that set variables to specific values. This enables a generic script to have a

different effect depending on which router, interface or PVC is being configured. For example, the local context for a Frame Relay interface could set values for the CIR, Bc and Be parameters, which could then be used later for Frame Relay Traffic Shaping, using a Sub-interface script applied to the network level.

---

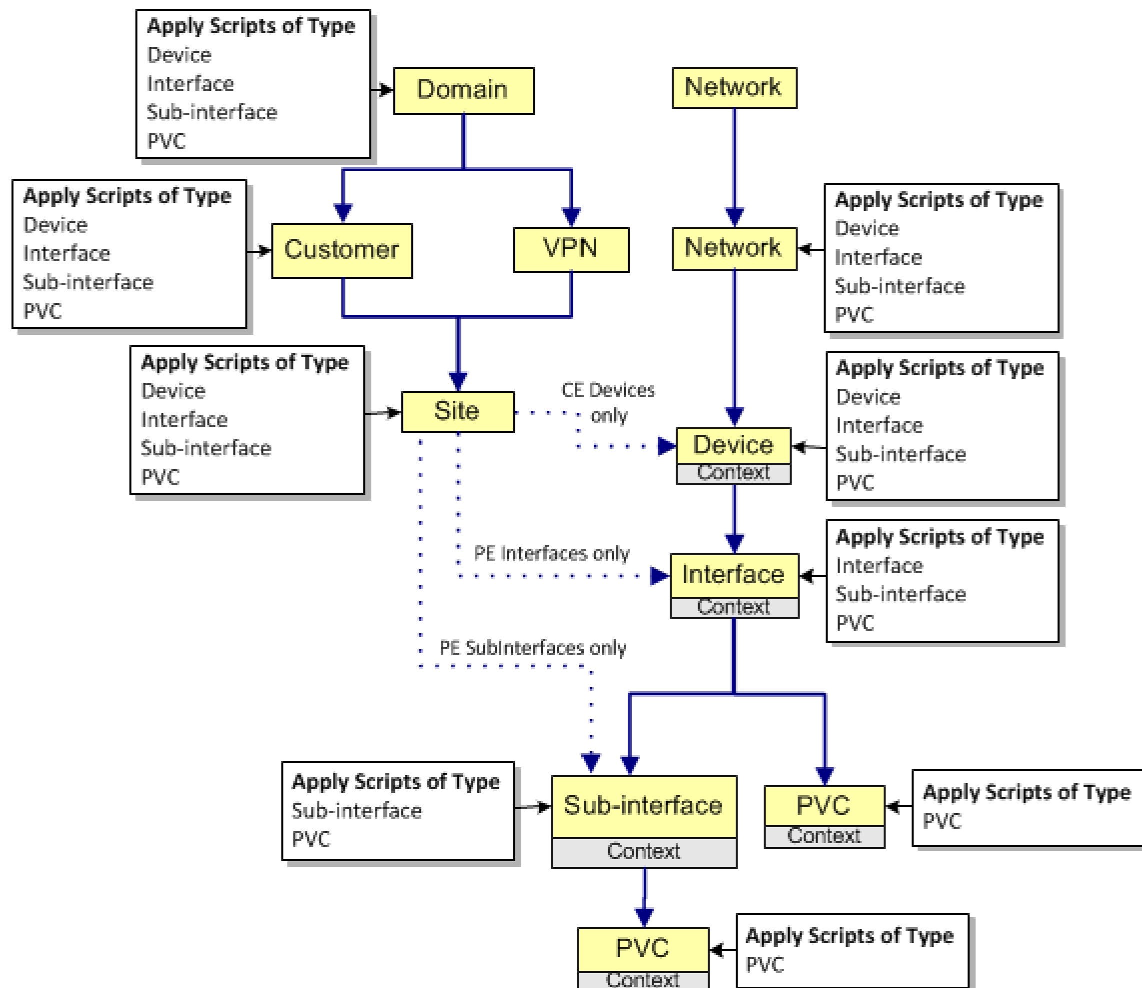
**Note:** A maximum of 512 characters can be included in an object's local context definition.

---

## Inherited Context

In addition to an object's local context, each object inherits specific context from its parent objects. For example, a PVC endpoint can potentially inherit context defined for its parent sub-interface device and network. Figure 2–1 shows the way in which driver scripts are inherited.

***Figure 2–1 Driver Script Inheritance***




# About Scheduling of Scripts

As for any other IP Service Activator configuration, driver scripts are implemented when a transaction is committed, resulting in configuration being propagated to network devices.

Users can define whether a driver script is run before or after any other configuration changes applied at the same time.

A script can be scheduled to run once, in which case it will be run when the transaction is committed, or repeatedly, in which case it will be run on the next commit and every subsequent commit until it is removed from the object.

When a script is scheduled to run once, users can specify to create concretes only once, on first execution of the script. When this attribute is set to true, it prevents re-running the script and recreating concretes after a driver restart, or after remanaging the device (unmanage then manage), or after role reassignment.

It is also possible to specify that a script is run when the system detects that a device has restarted. In this case, after the transaction containing the script application has been committed it will run whenever the device restarts, without the need for a further commit.

The Install section of a driver script is run when a script is associated with one or more objects.

The Remove section of a driver script is run when a script is set to disabled or unlinked from an object.

> **Note:** If a script is set to run once only, and not when a device is restarted, the Remove section can never be run as the device driver does not preserve the script. If a script is set to run once only and run on a restart, then it is possible for the Remove section to be run.

# About Running Scripts

A driver script can be applied to an object either by being directly linked or as a result of inheritance. The various sections of the script and its context (local and inherited) are combined to produce a Python script (that is, a concrete driver script object) that is run in the device driver against the selected device or a component of the device.

> **Note:** Scripts are not processed by the IP Service Activator policy server. This means that they are not checked or validated in any way.

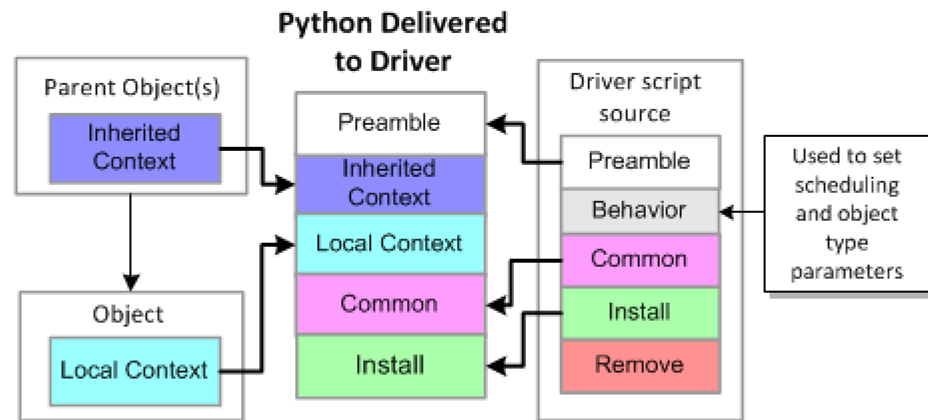
## Scripts to Install Configuration

When a script is to install configuration, it is constructed and run in the following order:

1. Preamble section
2. Inherited context
3. Local context
4. Common section
5. Install section

Figure 2–2 shows how a concrete driver script to apply configuration is constructed.

***Figure 2–2 Construction of a Concrete Driver Script to Apply Configuration***




Parameters defined in the local context always override those inherited from parent objects.

The information in the Behavior section does not form part of the executed concrete script, but is used to define where and when the script is run.

The script returns a result code, which will be OK unless appropriate tests are put in place as part of the script.

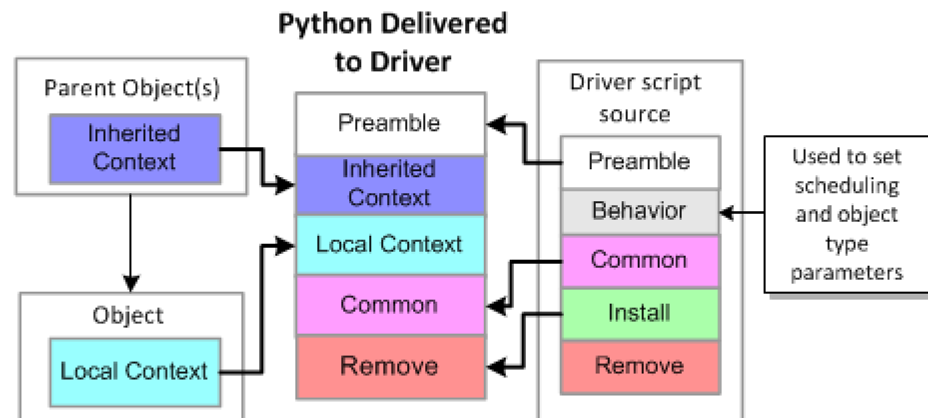
## Scripts to Remove Configuration

When a script is designed to remove configuration, it is constructed and run in the following order:

1. Preamble section
2. Inherited context
3. Local context
4. Common section
5. Remove section

Figure 2–3 shows how a concrete driver script to remove configuration is constructed.

***Figure 2–3 Construction of a Concrete Driver Script to Remove Configuration***



Parameters defined in the local context always override those inherited from parent objects.

The information in the Behavior section does not form part of the executed concrete script, but is used to define where and when the script is run.

The script returns a result code, which will be OK unless appropriate tests are put in place as part of the script.

## About Sharing Data Between Scripts

For each device that it manages, the driver allocates an area of memory that may be accessed by any Python scripts that are applied to that device. This is referred to as the shared data area or shared area. Information may be stored in this area as a Python dictionary or as a class-based structure. If a class-based structure is required, a Python module must be written that defines the data structure and passed to the device driver using a command-line option.

The shared data area supports the interchange of data between scripts and has a wide range of applications, for example, facilitating script ordering or optimizing re-use of basic functions used across scripts.

For more information, see "Sharing Data Between Scripts".

# 3
# Using Existing Scripts

This chapter explains how to manage existing driver scripts from within the Oracle Communications IP Service Activator user interface (client). It includes the following:

For details of requirements for running specific pre-defined scripts, see "Pre-defined Scripts".

## Importing Scripts

Any pre-defined scripts included with the system are installed in the **Service Activator\DriverScripts** directory. Before applying them you need to import the scripts you intend to use.

To import a script:

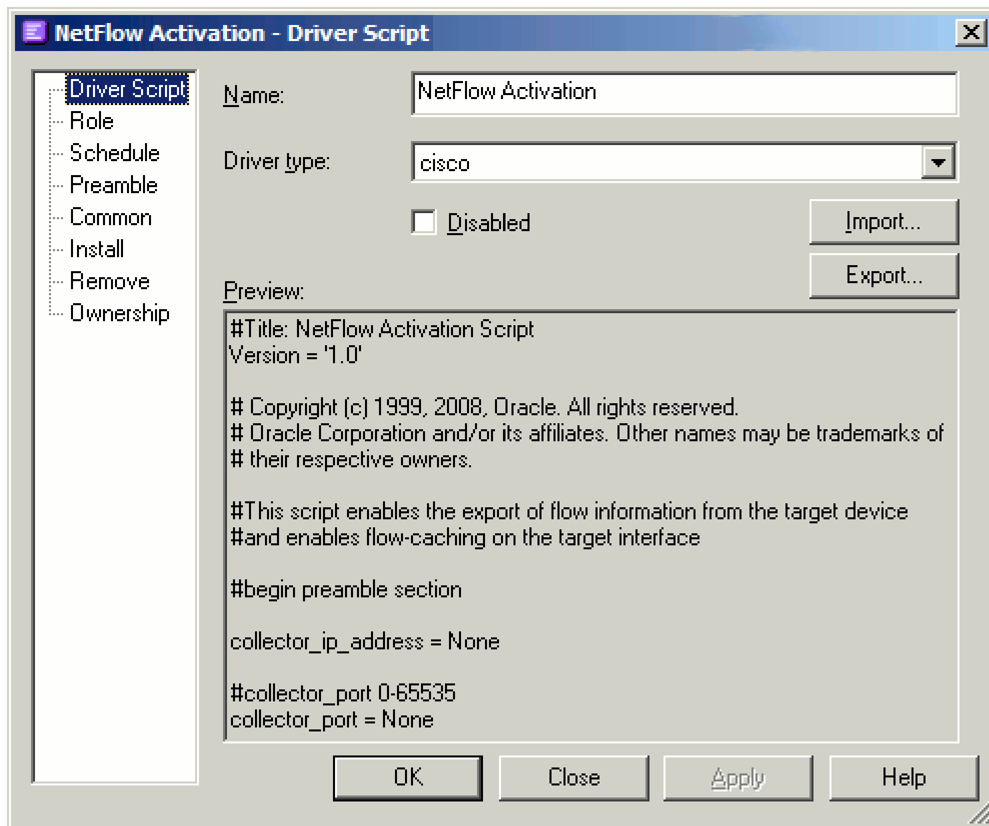
1. From the **Custom** tab, click the **Driver Scripts** folder.

   The **Custom** tab is available within the Global Setup window or any domain management window.

   A context menu appears.

2. Right-click and choose **Add Driver Script**. The Driver Script dialog box appears, as in Figure 3–1.

***Figure 3–1 The Driver Script Dialog Box***




3. Click **Import**. A file selection dialog box appears, listing all available scripts in the DriverScripts directory (.py files). Any Oracle-supplied scripts will appear in this list.
4. Choose the appropriate script, then click **Open**. The script is loaded.

The name of the imported script is set to the value of the *script_name* variable in the Behavior section, if present. Other variables in the Behavior section set details of device and interface roles (on the **Role** properties page) and scheduling requirements (on the **Schedule** properties page).

Scripts are not domain-specific. Once imported or created within a domain, a script is available within all domains.

---

**Note:** The order in which scripts are applied to a policy object is dependent on the order in which they are imported into IP Service Activator. Changing the list order of scripts in the client does not have any effect on their order of application. If script application order is critical, you can specify it using the shared data area. For more information, see "Sharing Data Between Scripts".

---

## Viewing and Organizing Scripts

Driver scripts are accessed from the **Custom** tab, which is available from both the Global Setup window and all domain management windows.

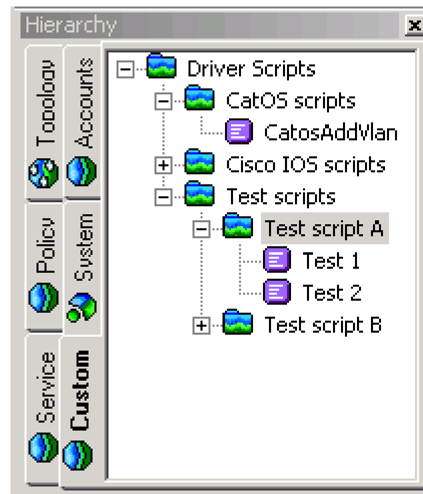
All scripts that have been imported or created within the client appear in the **Driver Scripts** folder. They can be organized into a structure of folders for administrative purposed.

## Creating Driver Script Folders

Within the **Driver Scripts** folder, you can create a folder hierarchy to organize the driver scripts. You can create multiple levels of folders, and you can drag scripts between folders. You can also set permissions on folders in order to control which users are able to access them.

Figure 3–2 shows an example hierarchy for the **Driver Scripts** folder.

***Figure 3–2 Driver Scripts Folder Hierarchy***




To create a driver script folder:

1. On the **Custom** tab, right-click on the **Driver Scripts** folder.

   A context menu appears.

2. Select **Add Folder**.

   The Driver Script dialog box appears.

3. Provide a name for the folder, and add remarks if necessary. Select the **Ownership** property page to set permissions on this folder.

   ---

   **Note:** Folders are not “groups” of driver scripts: it is not possible to apply a folder of scripts to a target

   ---

## Viewing a Summary of Scripts

To view a summary of the existing scripts:

1. Click on the **Driver Scripts** folder.

   The existing scripts will be listed in the **Details** pane, as in Figure 3–3.

*Figure 3–3 List of Existing Scripts in the Details Pane*

| Name | State | Level | Driver Type | Applies To | Device Role | Interface Role |
|---|---|---|---|---|---|---|
| Netflow Activation | | Custom | cisco | Interface | Any | Any |
| SAA DNS Connect | | Custom | cisco | Interface | Any | Any |
| SAA HTTP Web Server Re... | | Custom | cisco | Interface | Any | Any |
| SAA ICMP End-to-End acti... | | Custom | cisco | Interface | Any | Any |
| SAA Jitter Measurement | | Custom | cisco | Interface | Any | Any |
| SAA Responder | | Custom | cisco | Interface | Any | Any |
| Save to NVRAM | | Custom | cisco | Device | Any | Any |
| Script A | | Custom | cisco | ATM PVC | Any | Any |
| Script B | | Custom | cisco | Device | Any | Any |
| Script C | | Custom | cisco | Frame Relay PVC | Any | Any |

The **Details** pane provides the following information about each script:

- **Name**: name of the script
- **State**: indicates the current status of the script; values are:
  - **Inactive**: the script has been created, but not yet propagated to the devices
  - **Active**: the script has been propagated to proxy agents, but is not yet configured on a device. A once-only script returns to Active status once it has been run; you can run it again by reapplying it to the device
  - **Installed**: the script has been propagated to proxy agents and has been successfully installed on the designated device
  - **Failed**: the proxy agent experienced a failure trying to install the rule and it has therefore been discarded
  - **Rejected**: the script has been rejected by the device driver (for example, because of a syntax error)
- **Level**: indicates the level within the object hierarchy at which the script is applied (not relevant on this screen)
- **Driver Type**: the driver that will run the script
- **Applies To**: the target that the script applies to; values are Device, Interface, Sub-interface, ATM PVC or Frame Relay PVC.
- **Device Role**: the device role(s) that this script will be applied to.
- **Interface Role**: for Interface, sub-interface, ATM PVC and FR PVC scripts, the interface role(s) that this script will be applied to.
- **Installed**: indicates when this script will be applied relative to other configuration changes; value is either **Before config. changes** or **Following successful config. changes**.
- **Frequency**: indicates how often the configuration will be applied. Value is either **Once only** (that is, on the next commit) or **Repeat** (on each commit until the script is removed).
- **Install on restart**: indicates if the script is to be run when a device restart is detected; value is either **True** or **False**.
- **Owner**: if ownership of the script has been specified, value is the owner’s username.

- **Owner Group**: the group to which the owner belongs

## Viewing the Entire Text of Scripts

To view the entire text of a specific script:

1. On the **Hierarchy** pane, select the desired script.

   The following script text appears in the **Details** pane:

```
# Title: Netflow Activation Script
Version = '1.0'

# Copyright (c) 1999, 2008, Oracle. All rights reserved.
# Oracle Corporation and/or its affiliates. Other names may be trademarks of
# their respective owners.

#This script enables the export of flow information from the target device and
enables flow-catching on the target interface

#begin preamble section
```

Oracle recommends that you do not edit the script text directly in the details pane, although it is possible.

**Note:** The Behavior section of imported scripts is not visible in the Details pane.

## Viewing Script Properties

To view more details about a script:

1. Right-click on the script object.

   A context menu appears.

2. Select **Properties**.

   The Driver Script dialog box appears, as in Figure 3–1, "The Driver Script Dialog Box".

The Driver Script dialog box displays pages with the following information about the script:

- **Driver Script**: Displays identification information, including the first lines of the script.
- **Role**: Identifies the device role and interface role of the script. This information is defined in the Behavior section of the script; if necessary it can be edited on this page.
- **Schedule**: Identifies the scheduling requirements of the script. This information is defined in the Behavior section of the script; if necessary it can be edited on this page.
- **Preamble**: Displays the Preamble section of the script. Variables defined in this section normally have default values, which can be amended if required. Note that these values can be overridden by the context defined for particular objects.
- **Common**: Displays the Common section of the script. For existing scripts you are advised not to change anything on this page.

- **Install**: Displays the Install section of the script. For existing scripts you are advised not to change anything on this page
- **Remove**: Displays the Remove section of the script. For existing scripts you are advised not to change anything on this page.
- **Ownership**: Displays the permissions granted on the driver script. Only the owner of this object or a Super User can amend this page.

## Associating Roles with Scripts

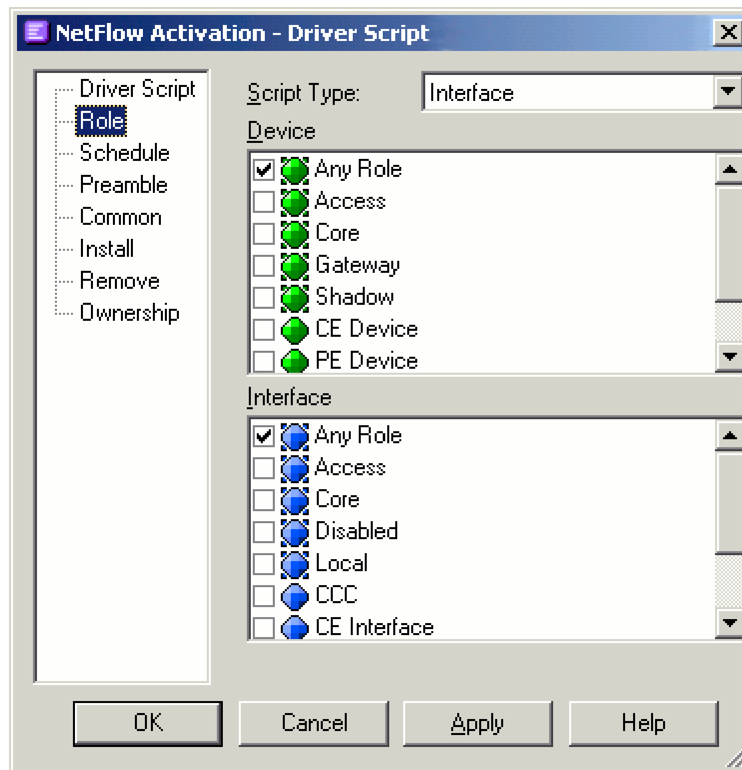
Before implementing a script, you need to apply the appropriate device roles, and, if necessary, interface roles, to define the points in the network to which the script applies.

To apply device or interface roles:

1. Display the Driver Script dialog box for the desired script. See "Viewing Script Properties".
2. Select the **Role** page.

   Figure 3–4 shows the Role page.

***Figure 3–4 The Role Page***




   The Script Type drop-down menu indicates the target of the script. This is defined as part of the script and should not be changed.

3. Select one system-defined device role (Core, Access, Gateway or Shadow) and one user-defined device role.

4. If the script type is Interface, Sub-interface, ATM PVC or Frame Relay PVC, you select one system-defined interface role (Local, Access, Core or Disabled) and one user-defined interface role.

The system-defined Any Role can be used to apply the script to any device or interface, whatever the role.

For more information about roles and how to assign roles to network elements, see *IP Service Activator Concepts* and *IP Service Activator User's Guide*.

## Setting Variables

The variables required when scripts are run can be defined in two places:

- In the Preamble section of the script, displayed on the Preamble page of the Driver Script dialog box. Variables set here apply to all instances of this script.
- In the local context for the object(s), defined on the Script Context property page for the object. Variables set here apply to any script run on that object.

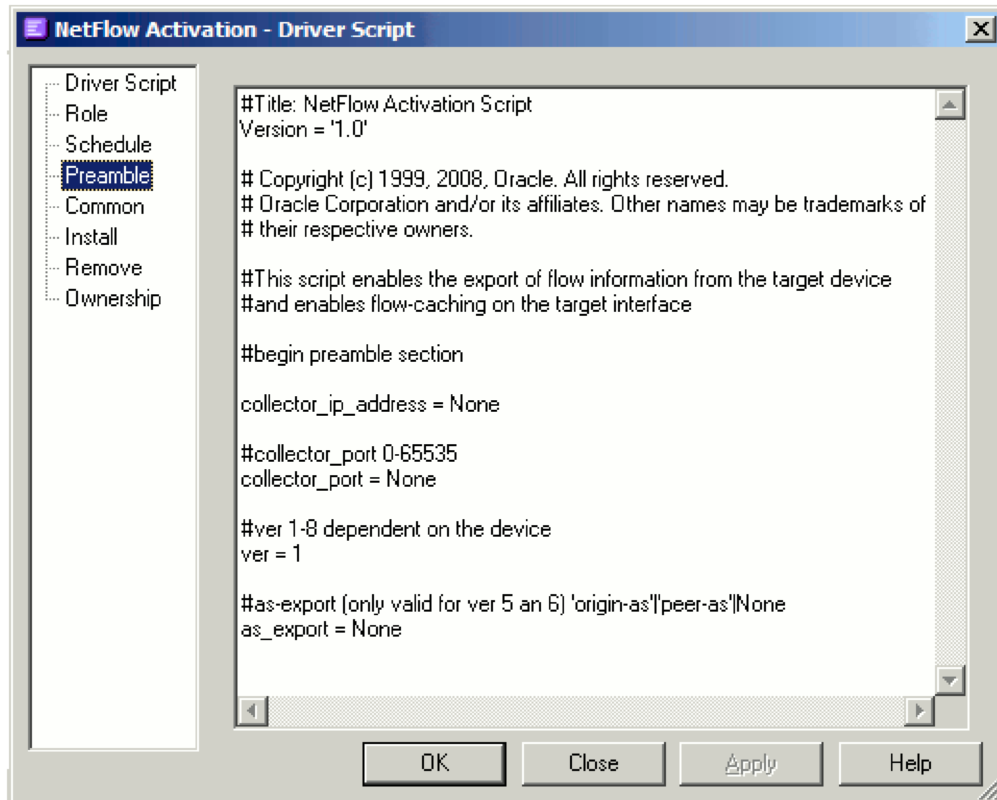
### Setting Variables in the Preamble Section

Variables normally have suitable default values supplied, but you can overwrite these if necessary with alternative values.

To set variables for a specific script:

1. Do one of the following:
   - Display the Driver Script dialog box for the selected script and select the Preamble page, as shown in Figure 3–5.

***Figure 3–5 The Preamble Page***




- Display the script in the Details pane

2. Edit the variables as required.

---

**Note:** Values set for parameters in the local context for an object override any values set for the same parameters in the Preamble section of a script.

---

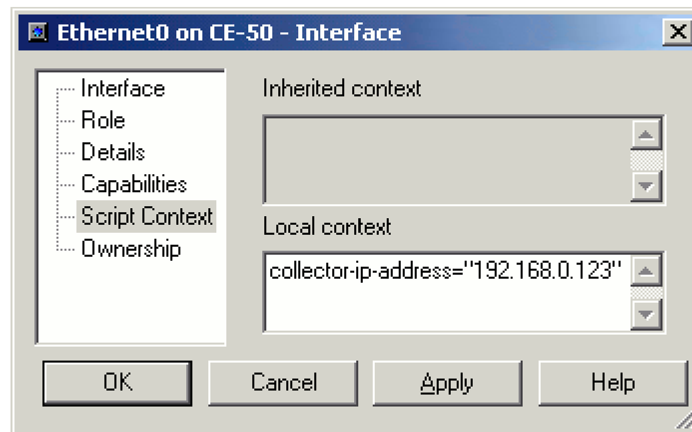
## Setting Variables in the Local Context for an Object

You can set variables that are specific to a particular object to which scripts can be applied. For additional guidance on managing script context, refer to "Managing Script Context".

To set variables for a specific object:

1. Select the object (such as device or interface) and display the Properties dialog box. Choose the **Script Context** page.
2. Enter the variables in the Local Context field, as shown in Figure 3–6.

*Figure 3–6 Script Context Page of Properties Dialog Box*




---

**Note:** A maximum of 512 characters can be included in an object's local context definition.

---

# Associating Scripts with Objects

Before a script is implemented, you must associate it with the appropriate network objects. Inheritance rules apply, so for example you can associate an interface script with a VPN and it will be applied to all relevant interfaces within that VPN.

## Linking Scripts to Objects

You link existing scripts to objects by dragging and dropping or by Copy and Paste commands.

To link a script using drag and drop:

1. With the appropriate destination object (such as a device or interface) displayed in the Details pane, drag the script object and drop it onto the destination object.

To link a script using Copy and Paste Link commands:

1. Select the script that you want to link, and choose **Copy** from the Edit menu (or click the Copy toolbar button).
2. Select the object to which you want to link the script, such as a VPN or a device, and choose **Paste Link**.

To see which scripts have been applied to an object you need to check the configuration details. See "Checking the Status of Scripts".

The script is not implemented on any network devices until the appropriate transaction is committed.

## Re-running a Script

A script set to run once can be run again if required.

To re-run a script on a particular object:

1. Select the relevant object and right-click on the script.
2. Select **Reapply Driver Script** from the context menu.

You must set the display so that both the relevant object and the driver script are visible – for example, by displaying the object in the Hierarchy pane and viewing its configuration in the Details pane.

To re-run all instances of a script:

1. On the Custom tab, right-click on the script from the **Driver Scripts** folder.
2. Select **Reapply Driver Script** from the context menu.

The script will be re-run on the relevant object(s) when the appropriate transaction is committed.

## Removing Scripts

To remove a script from a policy target:

1. Double-click on the relevant object to view the configuration applied to that object.
2. On the Configuration pane, select the **Driver Scripts** tab.
3. Right-click on the relevant driver script and select **Unlink** from the script's context menu.

If the script applies to any child objects by a process of inheritance, it will be removed from all objects in the hierarchy. Unlinking a script does not delete the script; it remains visible in the user interface and can be associated with objects later.

To remove all instances of a script from the network:

1. Right-click on the appropriate driver script and select **Properties**.
2. On the Driver Script page, select the **Disabled** checkbox.
3. Click **OK**.

The removal of the script takes place when configuration changes are next propagated to the network.

The actual action that occurs depends on the contents of the Remove section of the script. If no Remove section is defined, the script is run as requested but no action will result, that is, no configuration will be removed.

The Remove section is also run when a device is deleted. When a device is set to Unmanaged, the Remove section will be run if the Unmanaged action (set in the Options dialog box) is set to Remove.

---

**Note:** If a script is set to run once only, and not when a device is restarted, the Remove section can never be run as the device driver does not preserve the script. If a script is set to run once only and run on a restart, then it is possible for the Remove section to be run.

---

## Propagating Configuration

Once a script is linked to an object or unlinked from an object, it is implemented when the appropriate transaction is committed. When the transaction is committed any requested configuration is propagated from the policy server to the proxy agents and from there to the network devices.

### Applying Configuration

Where a script has been linked to an object, the commit procedure results in the Install section of the script being run on the specified object and any other object that has inherited the script.

The actual timing of the configuration changes will depend on the scheduling details specified on the Scheduling page of the Driver Scripts Properties dialog box:

- If **Before standard configuration changes** is selected, the script is applied before any other configuration performed at this time. If **After standard configuration changes** is selected, the script is applied immediately after any other configuration is applied.
- If **Once** is selected, the script is applied on the first configuration only. If **Repeatedly** is selected, the script is applied on all subsequent configuration changes until it is removed.
- If **Apply on Restart** is selected, the script is always applied when the system detects a device restart (in addition to before configuration is applied).

### Removing Configuration

The committed transaction results in the Remove section of the script being run in the following situations:

- When a script has been unlinked from an object
- When a device has been deleted
- When a script has been disabled
- When a device has been set to Unmanaged and the Unmanaged Action (set in the Options dialog box) is set to Remove (to specify that configuration is removed when a device is unmanaged).

> **Note:** Iif a script is set to run once only, and not when a device is restarted, the Remove section can never be run as the device driver does not preserve the script. If a script is set to run once only and run on a restart, then it is possible for the Remove section to be run.

The Remove section of the script is run on the relevant object and any other object that has inherited the script.

The scheduling settings have no effect when the Remove section is run.

> **Note:** The Remove section is optional. If no Remove section is defined in a script, the script is run as requested but no action will result. Oracle recommends that a Remove section is always included to ensure that configuration can be easily taken off.

## Deleting Scripts

You can delete scripts that are not linked to any other object.

If you delete a predefined script that is supplied by Oracle, it is removed from the user interface. If necessary you can import it again later.

If you uninstall IP Service Activator, all pre-defined scripts are deleted from the **DriverScripts** directory. User-defined scripts are not deleted. If you want to keep a copy of a pre-defined script, you should export it, saving it under a different name (see "Exporting Scripts").

# 4 Developing Scripts

This chapter explains how experienced Python developers can use the Oracle Communications IP Service Activator Configuration Development Kit to develop their own scripts.

## About Developing Scripts

Driver scripts are a powerful way of configuring your network. Please consider the following before creating scripts:

- The driver script feature is intended for experienced system developers only. You should only produce scripts if you are familiar with writing Python code.
- You must be familiar with the relevant device functions and commands that you intend to apply to the network.
- You should always test scripts thoroughly before applying them to devices.

For full documentation on Python, see the Python website at:

http://www.python.org

## Creating a Script

You can either create a script by using a text editor and importing the script into Oracle Communications IP Service Activator, or by entering the text directly within the IP Service Activator client.

### Using the Template File

To ensure that driver scripts are displayed correctly on the client and present a consistent approach, we recommend that a standard template is used when writing scripts. A suitable text file, **script_template.py**, is installed in the **DriverScripts** directory.

This template is as follows:

```
#Title: title
Version = "n.n"
#IP Service Activator version #.#.#
#Date: dd-mmm-yyyy
#Copyright statement
#<Short description of purpose and use of script, up to 3 lines>

#begin preamble section
#N/A
```

```
#begin behavior section
script_name = ""
script_driver_type = "cisco"
script_type = Device
script_device_role = All
script_interface_role = All
script_apply_when = After
script_repeat = False
script_apply_on_restart = False

#begin common section
#N/A

#begin install section

#begin remove section
#N/A
```

## Creating a Script Using a Text Editor

Using a text editor is the preferred method of producing driver scripts as it enables scripts to be written, compiled and checked outside IP Service Activator.

Note the following:

- Oracle recommends that you base the scripts you create on the template file outlined in "Using the Template File".
- To ensure consistency, follow the guidelines given in "Structure of a Script".
- Import the completed script into IP Service Activator, as described in "Importing Scripts".
- If necessary, you can edit the scripts from the client.

## Creating a Script from the IP Service Activator Client

The client provides only a basic text editor and does not provide any of the features available in more advanced code editing software. We recommend that you create driver scripts (apart from very simple scripts), using other third-party software and import them using the driver script import feature as described in "Importing Scripts".

If you create a script from within the client, you can either create a standalone script (that is, not initially linked to any object) or a script that is directly associated with an object.

To create a standalone script:

1. From the Custom tab, select the **Driver Scripts** folder, or a folder within it.
2. Right-click and choose **Add Driver Scripts** from the context menu.
3. On the Driver Script page, enter a name for the script in the **Name** box, and select the driver type from the drop-down list.
4. On the Role page, choose the target objects that the script will configure. For all scripts, you must select the device role to which the script is to apply. You can choose one system-defined device role (Core, Access, Gateway or Shadow) and one user-defined device role. Choosing **Any Role** applies the script to any device assigned a role.

For Interface, Sub-interface, ATM PVC and Frame PVC scripts, select the interface role to which the script is to apply. You can choose one system-defined interface role (Local, Access, Core or Disabled) and one user-defined interface role. Choosing **Any Role** applies the script to any interface assigned a role.

If roles are not defined, the script will not be implemented.

5. On the Schedule page, specify when the script is to be run: before or after standard configuration changes, once or repeatedly, and whether or not it is to be run when a device is restarted. When you specify to run the driver script once only, you can further specify to create concretes once only, on first execution.
6. Enter the lines of the script directly on the Preamble, Common, Install and Remove pages, following the guidelines described in "Structure of a Script".

It is also possible to create a script for a particular object, for example, to create a script specific to a single device.

To create a script for a specific object:

1. Select the appropriate network object and choose **Add Driver Script** from the context menu.
2. Enter the details of the script on the Driver Script dialog box property pages, as described in "To create a standalone script:".

The script is automatically linked to the selected network object, and will be installed when the appropriate transaction is run.

The script also appears in the **Driver Scripts** folder on the Custom tab, allowing it to be easily re-used.

## Structure of a Script

A driver script consists of the following sections:

- The Preamble Section identifies the script and defines any variables used.
- The Behavior Section defines standard values that define the objects that the script applies to and the scheduling parameters..
- The Common Section contains common functions that can be called from both the Install and Remove sections. This section is optional.
- The Install Section consists of Python code that is only run when a script is installed on a device. Install code is mandatory.
- The Remove Section consists of Python code that is only run when a script is removed from a device. Remove code is optional.

In the source file, the different sections are delimited by comments within the Python script. In raw text format, the script therefore has the following format:

```
#begin preamble section
Preamble header
Python code for preamble section
#begin behavior section
Behavior parameter definitions
#begin common section
Python code for common section
#begin install section
Python code for install section
#begin remove section
Python code for remove section
```

Even if a section does not include any code, it is important to include these comment lines to ensure that when the script file is imported the script works properly and is displayed correctly on the Driver Script dialog box in the client. The comment line at the start of each section must be followed by a new line (CRLF), and there must not be a space or any other character after the word “section”.

If you export a driver script from the client to a file, the sections will be combined in the correct order and the relevant delimiters inserted.

The rest of this section defines the information that you should include in each of the five sections of a script.

## The Preamble Section

The preamble section is used to identify the script and define any variables used. It is Python code, and by convention should have the following format:

```
#Title: <Script Name>
Version = "<Version>"
#IP Service Activator version: #.#.#
#Date: <Date>
#Copyright (c) <Company> <Year>
#<Short description of purpose and use of script, up to 3 lines>
#
#
#begin preamble section
default definitions of any variables used in the script from the object context
```

The first eight lines of the Preamble section appear on the Driver Script page of the Script dialog box in the read-only Preview field. By convention these lines should therefore contain only the header comments. These lines must always be as follows:

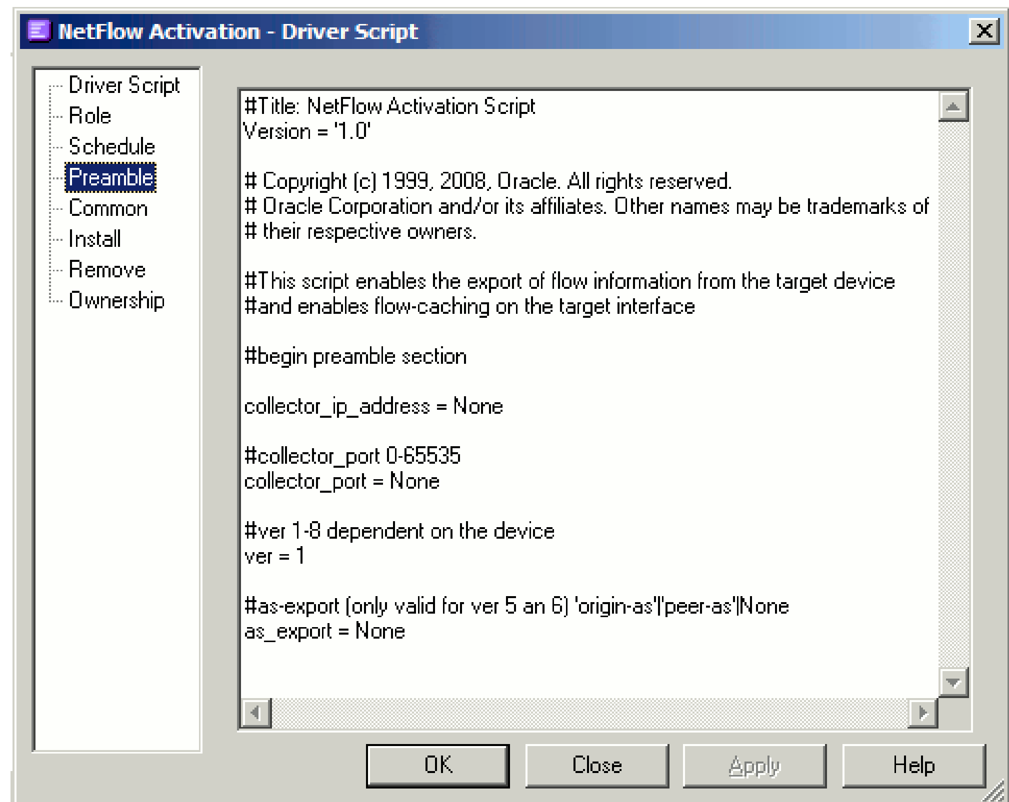
- Line 1 provides a short title for the script.
- Line 2 sets the Version variable, which identifies the version of the script. The version umber must be amended whenever the script is updated.
- Line 3 specifies the IP Service Activator release for which this script was written. Scripts may need to be amended for different releases.
- Line 4 lists the date that the script was last amended in dd-mmm-yyyy format.
- Line 5 provides the copyright statement, if applicable.
- Lines 6 to 8 explain the script. It is useful to limit the width of text so that it displays correctly in the client.
- Line 9 must always include the comment `#begin preamble section`. Unlike the other section comments, this line is not used as a section delimiter, but indicates the start of the variable definition section.

If there is no preamble, then the comment `#N/A` should appear on the next line.

If the script uses variables that are expected to be set in the context of objects to which the script is applied, then by convention the preamble section should set each to a default value, one per line.

On the IP Service Activator client, the Preamble section can be viewed (and edited if necessary) on the **Preamble** page of the Driver Script dialog box as in Figure 4–1.

*Figure 4–1 The Preamble Page*




## The Behavior Section

The Behavior section of a script is a set of variable definitions that control when and how the script is run. When a script is imported, this section is parsed and the various fields on the **Role** and **Schedule** pages of the Driver Script dialog box are set. When a script is exported to a file, the current field settings are written out in the definitions. If a Behavior section is not specified in a script, default values are applied.

Table 4–1 displays the variables that can be set in the Behavior section. The variables are case-sensitive.

The Behavior section cannot be viewed directly on a property page, but the settings can be viewed (and edited if necessary) on the **Driver Script**, **Role**, and **Schedule** property pages of the Driver Script dialog box.

*Table 4–1 Variables Set in the Behavior Section*

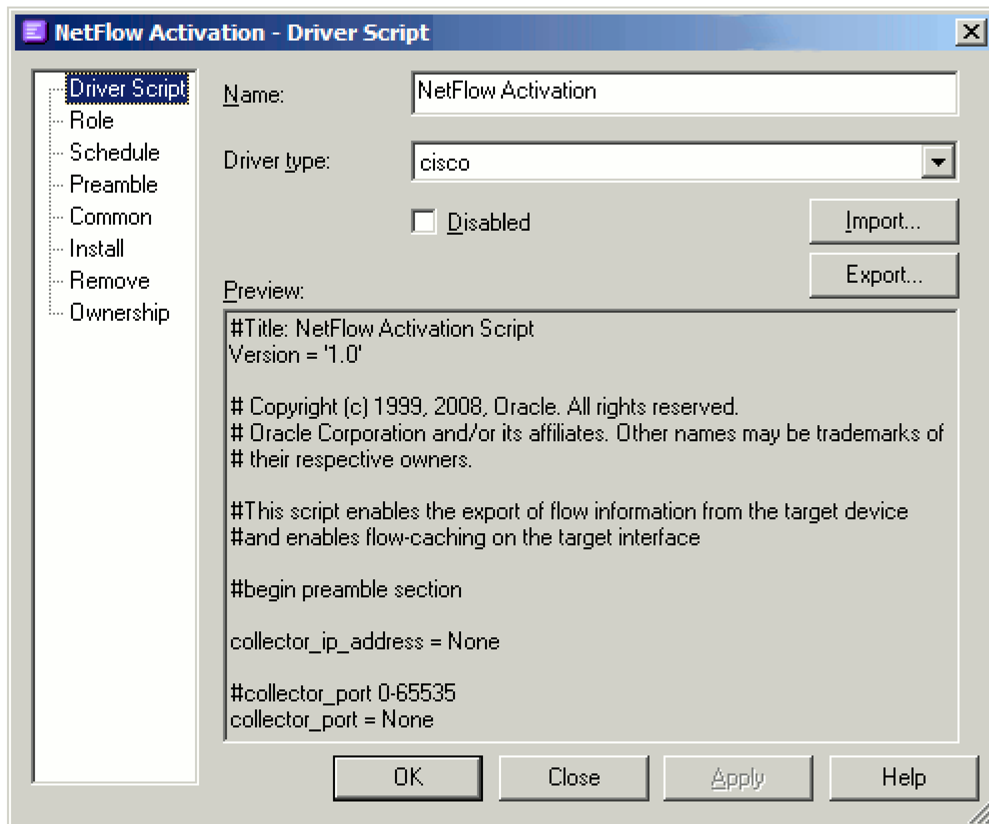
| Variable | Property Page | Purpose | Default |
|---|---|---|---|
| script_name | Driver Script | String; the name of the script. Should be the same as the title defined on line 1 of the script. | " " |
| script_driver_type | Driver Script | String identifying device driver. This must match the name of the appropriate driver component. | "cisco" |
| script_type | Role | Must be one of **Device**, **Interface**, **Subinterface**, **Atmpvc**, or **Frpvc** | Device |
| script_device_role | Role | Must be one of **Access**, **Gateway**, **Core**, **Shadow**, or **All**. | All |

*Table 4–1 (Cont.) Variables Set in the Behavior Section*

| Variable | Property Page | Purpose | Default |
|---|---|---|---|
| script_interface_role | Role | Must be one of **Access**, **Core**, **Local**, **Disabled**, or **All**. | All |
| script_apply_when | Schedule | Either **Before** (script is run before other configuration) or **After** (script is run after other configuration). | Before |
| script_repeat | Schedule | Either **True** (script is run on every subsequent command) or **False** (script is run once only, on next commit). | False |
| script_create_ concretes_once | Schedule | Either **True** (concretes are created only on the first successful run) or **False** (concretes are created on every run). | False |
| script_apply_on_ restart | Schedule | Either **True** (script is run whenever a device restart is detected) or **False** (script is not run on a device restart). | False |

For example, Figure 4–2 shows the setting of *script_name* and *script_driver_type* in the **Driver Script** page.

*Figure 4–2 Setting Variables on the Driver Script Page*




## The Common Section

The Common section is Python code available whenever the script is run. It is used to define functions that can be called from both the Install and Remove sections.

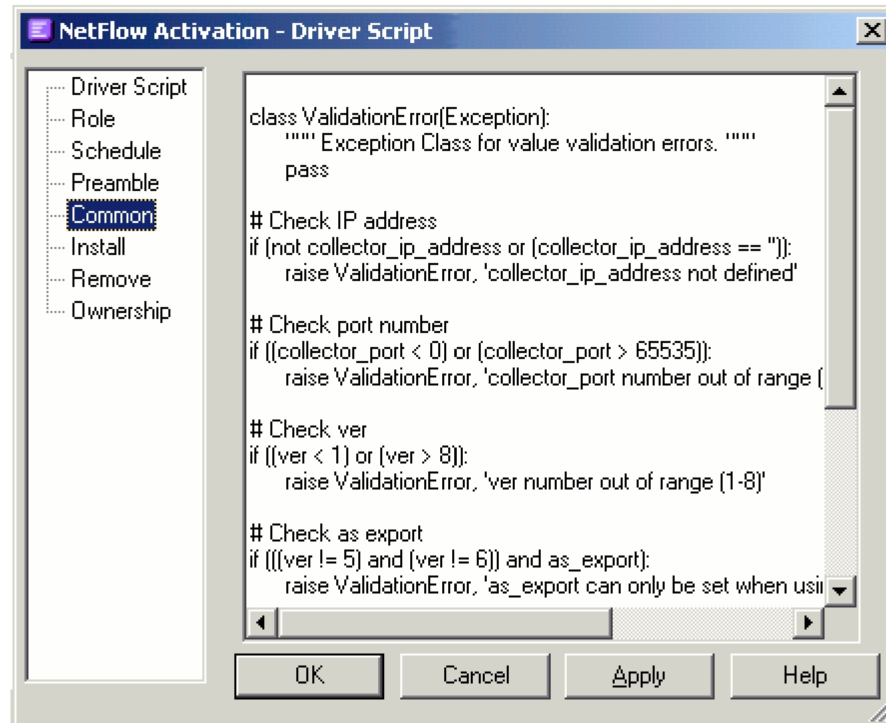
By convention, the Common section should start with the comment:

```
#begin common section
```

The Common section is optional; if you do not need to define any common functions, the comment `#N/A` should appear on the next line.

On the IP Service Activator client, the Common section can be viewed (and edited if necessary) on the **Common** page of the Driver Script dialog box, as shown in Figure 4–3.

***Figure 4–3 The Common Page***




## The Install Section

The Install section is Python code that is run when the driver script is associated with an object. This will be at the first propagate after the script is associated with the object and optionally on each subsequent propagate or device restart (for details, see "About Scheduling of Scripts").

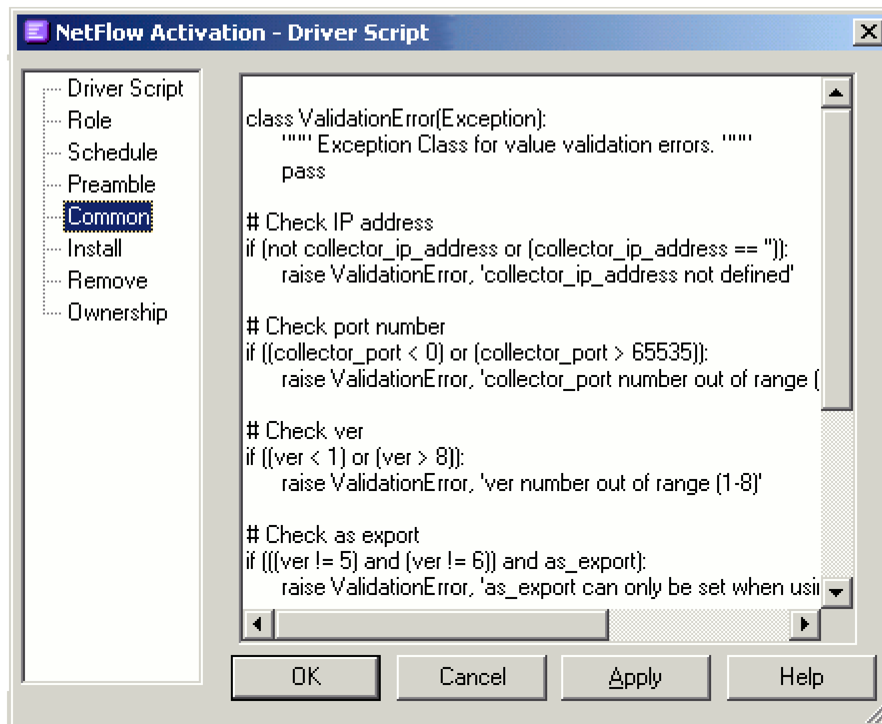
By convention, the Install section should start with the comment

```
#begin install section
```

**Note:** An Install section is mandatory.

On the IP Service Activator client, the Install section can be viewed (and edited if necessary) on the **Install** page of the Driver Script dialog box, as shown in Figure 4–4.

*Figure 4–4 The Install Page*




## The Remove Section

The Remove section is Python code that is run when the association between the driver script and an object is removed, either by unlinking the script from an object or by deleting the script. Its purpose is to remove any configuration that was installed by the Install section.

The Remove section is not mandatory, but for a script that installs configuration on devices, Oracle strongly recommends that you define code that removes the configuration that has been placed by the Install code.

---

**Note:** If a script is set to run once only, and is not set to run when a device is restarted, the Remove section can never be run as the device driver does not preserve the script. If the script is set to run on a restart, then the Remove section can be run.

---

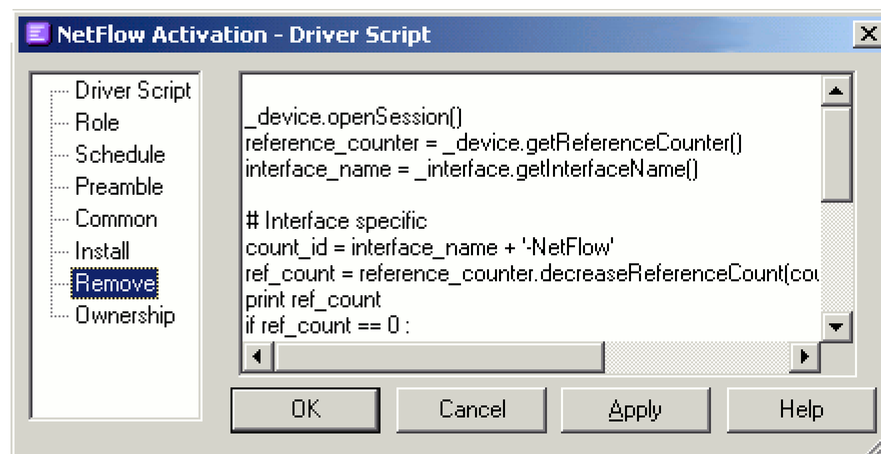
By convention, the Remove section should start with the comment:

```
#begin remove section
```

If you do not need to define a Remove section, the comment `#N/A` should appear on the next line.

On the IP Service Activator client, the Remove section can be viewed (and edited if necessary) on the **Remove** page of the Driver Script dialog box, as shown in Figure 4–5.

***Figure 4–5 The Remove Page***




For examples of actual scripts, see "Pre-defined Scripts".

# Exporting Scripts

You can export any scripts created or amended within the client. This allows you to save an existing script under a different name.

To export a script:

1. On the Custom tab, within the **DriverScripts** folder, select the script that you have created or amended within the client.
2. Choose **Properties** from the context menu.
3. Click **Export**. A standard file selection window appears.
4. Specify the filename for the Python script. By default, files are saved in the **DriverScripts** folder; you can choose another location if you wish.
5. Click **Save**.

The exported script file can be viewed and edited using a text editor. The exported file will include a Behavior section, created from the attributes set on the Driver Script, Role and Schedule property pages.

---

**Note:** When a script is exported, no user-defined roles are saved within the exported script.

---

# Programming Tips

This section gives tips to help when developing scripts.

## Script Conventions

When writing scripts, do not define variables starting with an underscore character as this is restricted to variables defined by IP Service Activator, such as **_result** and **_device**.

For general discussions about Python programming style, see the Python documentation index at:

http://www.python.org/doc/essays/

## Command Format

When applying commands to a router, use the full form of all commands rather than abbreviated commands. As well as helping to ensure compatibility with different operating system versions, using full commands makes the script more understandable.

## Handling Exceptions

The device driver explicitly catches all errors and returns a status of `Failed`, together with the error and line number. For more fine-grained error handling, use multiple Try/Except blocks for specific exceptions.

In the following example, a call is made to an exception handler defined in the Common section.

```
#begin common section
def handleException(details):
  _result.setCode(_result.FAILED)
  _result.setDetails(details)

#begin install section
try:
  _device.openSession()
except RuntimeError, err:
  handleException(err)
  return
```

## Displaying Errors in IP Service Activator

Because IP Service Activator can display only one line of text in an error message, when diagnosing script errors it can be useful to copy the text of the error into a text editor, which can display multiple lines including end-of-line characters.

To copy the error text into a text editor:

1. Open the Device Configuration Log.
2. Locate the relevant CDK error.
3. Right-click on the error and select **Copy**.
4. Open the text editor.
5. Select **Paste**.

The error is then displayed correctly formatted.

## Applying Commands

If the script is to apply commands to a router by calling **deliverCommand('')**, then it must first call **_device.openSession()** and finish by calling **_device.closeSession()**.

For example:

```
_device.openSession()
_device.deliverCommand('copy running-config startup-config')
_device.closeSession()
```

It is good practice to validate all the values that can be set by users, since if a value is set incorrectly no command is delivered. Oracle recommends including validation checks in the Common section of a script to ensure that no configuration is installed until all the values have been checked. Checking each command in turn in the Install section can result in the router being left in a partially-configured state if one command fails.

## Preventing Command Application

The **-NoCommandDelivery** flag on the Cisco driver typically prevents the driver from sending a configuration to a device, sending it to the audit log files (pre-pended by `no-command-delivery`) instead.

Commands sent by scripts are not executed when the driver is in NoCommandDelivery mode.

An optional second parameter to **deliverCommand** lets script writers identify commands as read or write. This is useful when developing scripts with the driver in NoCommandDelivery mode to prevent accidental changes.

Read commands should not modify the device and so are valid to send when not delivering commands. This is similar to the behavior of the device driver itself which will send show commands regardless of state. (While this is the intent, it is on the honor system. IP Service Activator does not try to validate whether or not the command will actually modify the device.)

The command access can be specified with `_device.deliverCommand(command. _device.read)` or `_device.deliverCommand(command, _device.write)`

A read command is always executed. A write command is logged in the audit log with a prefix of `CDK-no-command-delivery` if the driver is not delivering commands and is executed normally otherwise.

Script writers can check if write commands will be delivered:

`_device.deliveringCommands()` returns true or false to reflect the driver state.

They can also check the result of **deliverCommand** which is always `NoCommandDelivery enabled` if the driver is not delivering commands.

## Processing Command Output

If you want to get and process the output of an operating system command, then you need to store the result of **deliverCommand('')**.

For example:

```
Result = _device.deliverCommand('show running-config')
```

You can then use normal Python techniques to extract the information you need.

## Returning a Result

By default the return code will be `TRUE`. You need to write the script to catch exceptions and deal with any other errors by returning `FAILED`. You can also provide a detail string which is logged by the server.

For example:

```
_result.setCode(_result.OK)
_result.setDetails('An error has occurred')
```

## Re-applying Configuration

When writing a script, bear in mind that the configuration you wish to place on the device might already be there. If possible you therefore need to write a script that is intelligent enough to check for and deal with configuration that is already applied.

## Managing Script Context

Any variable created in Python outside the scope of a Python class or method is considered a global variable. This fact can cause a variable that is only applicable to the current script to be misapplied to subsequent scripts. This section describes a method to ensure that local variables are applied to the local context only.

The recommended method is to wrap any CDK script in a function definition before execution. In this way, all classes, functions and variables defined will be local to the wrapper function.

For example, you can wrap script context code in a class definition, as follows:

```
class ScriptContext:
  x=5
  y=6
  z=7
```

and then in the CDK script you would reference those variables as follows:

```
ScriptContext.x
ScriptContext.y
ScriptContext.z
```

A Python script is typically formatted as follows:

```
import DriverScript
import OrchestreamExceptions
import traceback
_scriptInfo = "user's cdk script (including script context)"

try:
     exec(_scriptInfo.getScript())\n"
except:\n"
     traceback.print_exc(file = _scriptInfo)\n"
```

To manage script context, the Python script should be formatted as follows:

```
import DriverScript
import OrchestreamExceptions
import traceback

def runPython():
    _scriptInfo = "user's cdk script (including script context)"
    try:
         exec(_scriptInfo.getScript())\n"
    except:\n"
         traceback.print_exc(file = _scriptInfo)\n"

runPython()
```

# 5

# Sharing Data Between Scripts

This chapter, aimed at experienced Python programmers, explains how to exchange data between scripts using the shared data area.

## About Sharing Data Between Scripts

It is possible to share data between Configuration Development Kit (CDK) scripts. This feature has a range of applications, including the ability to store and re-use data or functions that are commonly used by scripts and to exercise greater control over the order of script execution.

The device driver allocates an area of memory for each device that it manages. This is referred to as the shared data area or shared area. Shared data may be stored for the lifetime of the device, that is, until the device is unmanaged or the device driver restarted, or for a single transaction. Information that is currently held in the shared data area may be accessed by any CDK scripts that are applied to that device.

By default, data is stored in the shared area as a simple Python dictionary but it is also possible to structure the shared information using classes. If you wish to use a class-based structure, you must create a Python module that defines the structure and pass it to the device driver using a command-line option.

The class-based solution provides a more manageable approach to using the shared data area. It allows some initial setup to be performed by the Python module so that the shared data area contains useful data when the first CDK script is run against a device. This contrasts with the dictionary-based structure, where the dictionary is empty when the first CDK script runs and any data, classes or functions must be added by the CDK scripts themselves.

This chapter describes how to store information in the shared data area using classes or a dictionary, provides an example approach to script organization to make use of the shared area and outlines some potential applications.

## Using Classes

Storing data in the shared data area using a class-based data structure provides a maintainable use of the area.

The structure of the classes that will be stored in the shared data area must be defined as a Python module. A module is a file containing code which defines a group of Python functions or other objects.

The module is run when the device driver is started with the following command-line parameter:

```
-SharedDataModule module
```

where module is the name of the Python module that defines the structure of information held in the shared data area.

For example:

```
\Program\cisco_device_driver.exe|"-SharedDataModule CreateSharedArea …
```

The Python module must be installed on every machine that is hosting a device driver used to configure scripts.

---

**Note:** When specifying the name of the Python module, you do not need to include the script's.py extension.

---

If you are running the device driver on Solaris, the location of the Python module must be set in the PYTHONPATH environment variable.

If you are running the device driver on Windows, the driver looks for the module in **c:\Program Files\Oracle Communications\IP Service Activator\Program** by default. The PYTHONPATH variable may be set to an alternative location.

The module defines the basic structure of the object class stored in the shared data area and the methods available to instances of the class.

The module is a user-defined script and is likely to vary across installations according to the needs of the operation. At minimum, it should provide methods for inserting new objects in and removing objects from the shared area. A sample module is provided in "Sample Scripts for Using the Shared Data Area".

---

**Note:** The name of the function that the device driver calls to create the class-based structure is hard-coded within the driver. User-defined modules must contain a CreateSharedObject function. For example:

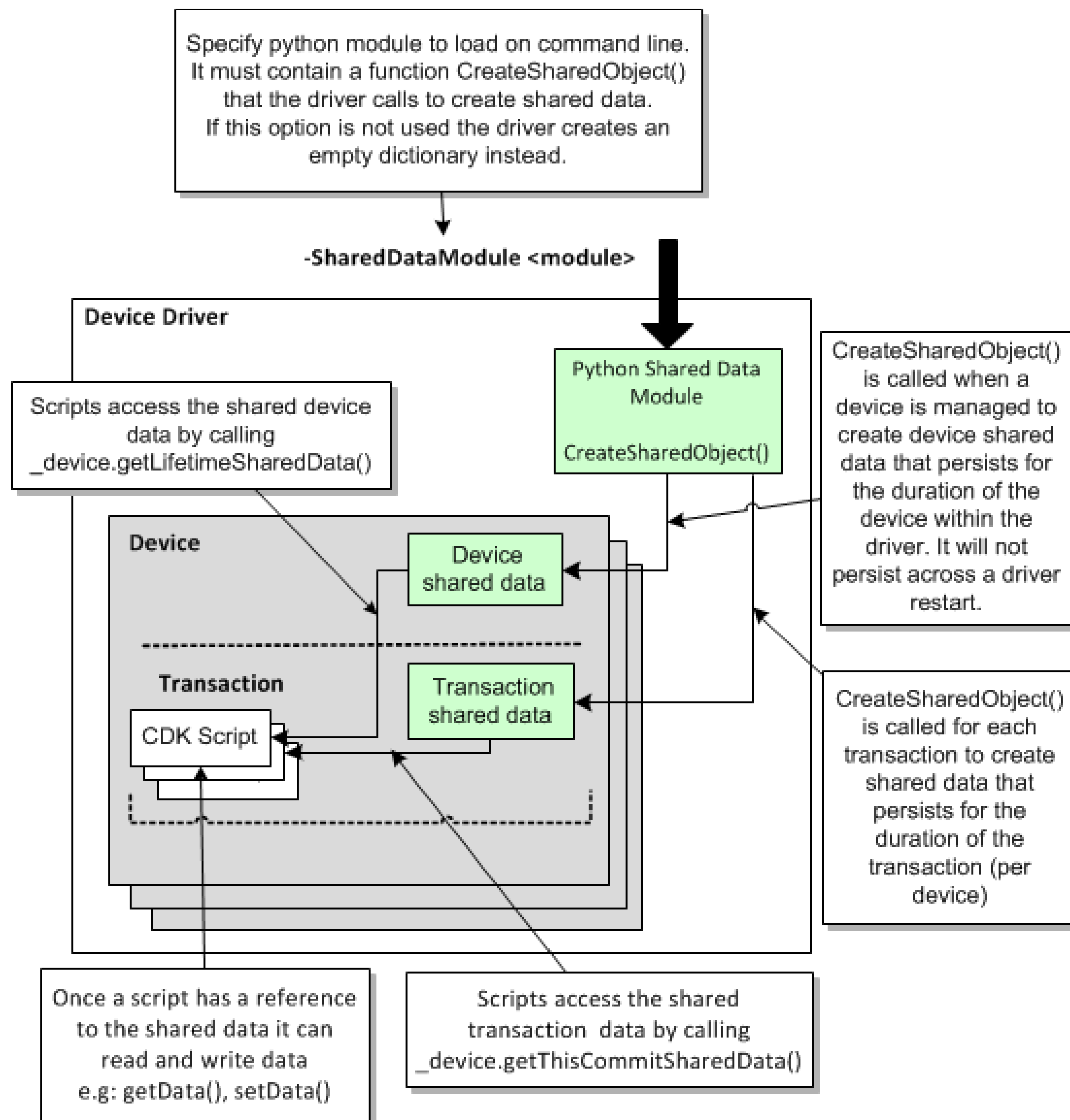
```
def CreateSharedObject():
return SharedInfo()
```

If there is no CreateSharedObject function, the driver reverts to a dictionary.

---

Figure 5–1 illustrates the class-based shared data area.

*Figure 5–1 The Class-based Shared Data Area*




## Using a Python Dictionary

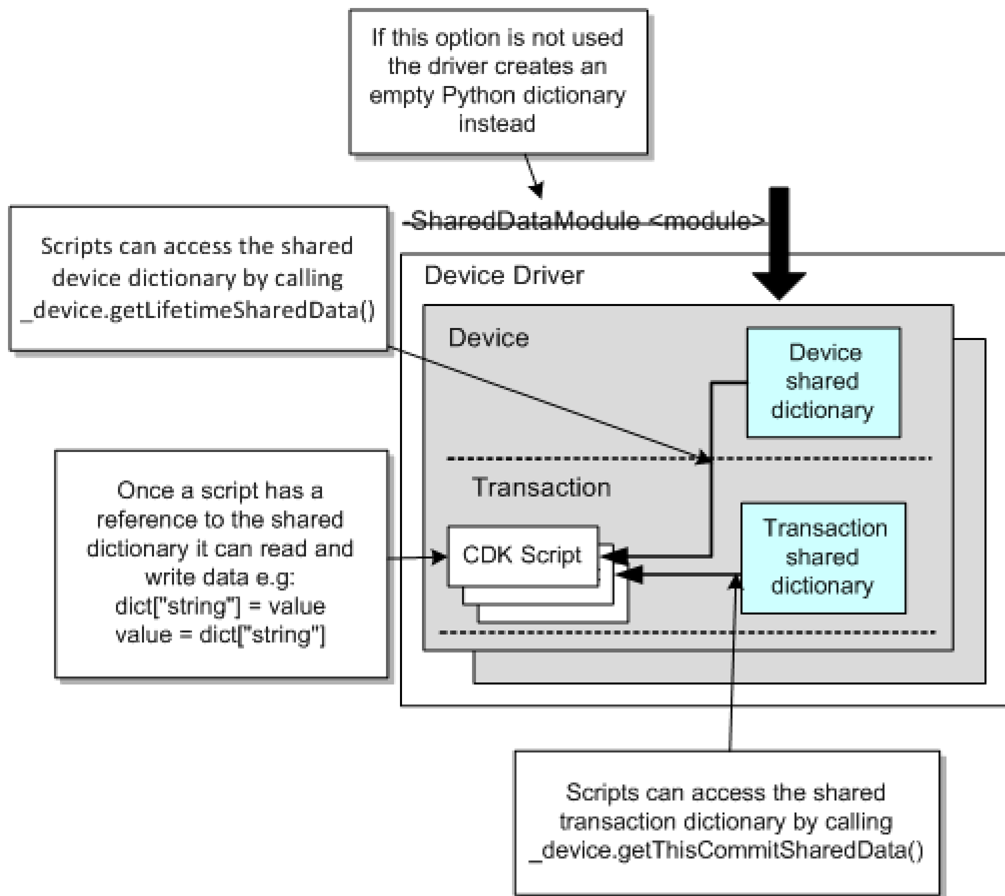
If the -SharedDataModule command-line parameter is not passed to the relevant device driver, the driver creates an empty dictionary in the shared data area on startup.

A Python dictionary is an associative array, implemented using hash tables, which provides access to values by integers, strings or other Python objects referred to as keys. A key indicates where in the dictionary a given value may be found. Because the key into a dictionary may be something other than a number, dictionary objects do not have any implicit ordering relative to each other. However, an explicit ordering may be defined using another data structure, such as a list. The objects stored in a dictionary may be of any type.

Once created, new positions may be created in a dictionary as values are written to them, without the need to create the position beforehand. Values stored in a dictionary may be accessed and used by scripts running on the same host machine.

Figure 5–2 illustrates the dictionary-based shared data area.

***Figure 5–2 The Dictionary-based Shared Data Area***




## Storing and Retrieving Data

The device driver creates two stores of shared data for a device: one that persists for the lifetime of the device, and one that persists for the lifetime of a transaction.

The following method retrieves the data that has been stored in the shared data area during the current transaction only:

```
_device.getThisCommitSharedData()
```

The following method retrieves data that is stored in the shared data area for the lifetime of the device:

```
_device.getLifetimeSharedData()
```

Once a script has a reference to the shared data it can read and write data to the area.

The technique for storing information in the shared area depends on whether data is stored as a class or dictionary.

If shared data is stored as a class object, it is possible to call the methods defined for the class. These methods are defined in the Python module named in the -SharedDataModule parameter on the device driver’s command line.

For example, assuming the existence of an insertBehavior() method for the shared data class:

```
_device.getThisCommitSharedData().insertBehavior(BehaviorX())
```

inserts an object of type BehaviorX in the shared data area.

If data is stored in a dictionary, it is possible to access data via its key. For example:

```
Address=_device.getThisCommitSharedData()["IPAddress"]
```

## An Example of Using the Shared Data Area

The CDK offers a very flexible solution to sharing data between scripts, enabling storage of data per device for the lifetime of the device, or for the lifetime of a transaction, with no restriction on the type of data to be stored or the structure used to organize it. Any use of the shared data area can therefore be geared towards the particular needs of the operation.

The example in this section illustrates the type of processing that may be performed using the shared data area and uses a dictionary-based approach to storing data. This means that all classes and methods are defined within the scripts themselves. If a class-based approach were used, these would be defined in a Python module passed to the device driver on the command line. This would simplify the data, behavior and controller scripts applied to devices. A sample Python module definition is provided in "Sample Scripts for Using the Shared Data Area".

### Types of Script

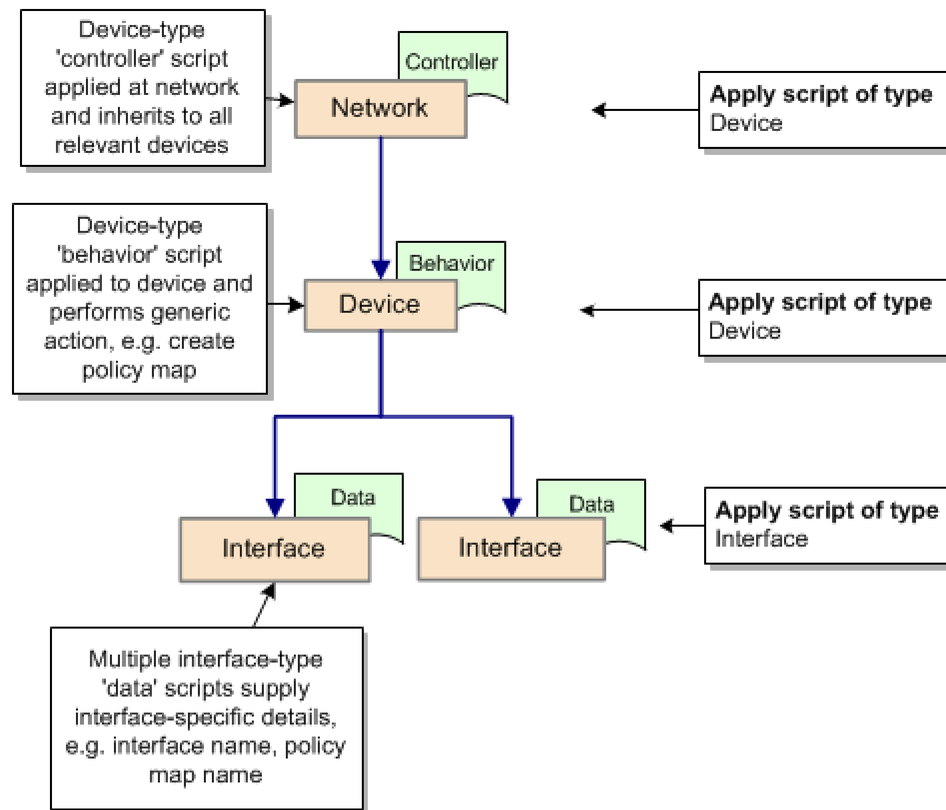
In order to support re-use of script functionality and reduce redundancy, the example described in this section divides scripts into three basic types, according to the type of function they perform:

- **Data scripts** define configuration data that is specific to a particular policy target, such as an interface.
- **Behavior scripts** define generic functions that can be run against multiple policy objects, for example, defining policy or class maps, or performing a generic operation such as parsing or ordering data. When run, a behavior script creates a behavior object in the shared data area.
- The **controller script** manages the execution of the behavior and data script objects (created by behavior and data scripts), for example, by prioritizing the order in which behavior and data objects are processed.

> **Note:** The division of scripts into behavior, data and controller scripts is designed to take full advantage of the features offered by the shared data area. The scheme presented in this example is a sample categorization and acts as an illustration only. You may choose to categorize scripts according to any other logical scheme.

The division of scripts into behavior, data and controller types optimizes re-use of the functionality defined by a script. For example, a behavior script can be applied to any number of devices to define generic aspects of policy map configuration and inherited to the relevant interfaces. A number of separate data scripts define the interface-specific information and are applied to each relevant interface. This is illustrated in Figure 5–3.

***Figure 5–3 Division of Scripts into Types***




## About the Example

The example shows how to apply configuration changes to a device, making a simple change to the interface IP address and description. Although the changes it makes are simple, it illustrates some basic principles and tasks that you may wish to use in your own scripts. These include:

- Parsing the device's running configuration and storing it in the shared data area.

  The example demonstrates how to retrieve device configuration only once and store it in the shared data area. The stored configuration is referenced by other scripts that are set to run against the device. This minimizes communication with the device and cuts down network traffic.

- Comparing potential new device configuration to that which is already installed

  This ensures that there is no unnecessary transmission of data to the device if, for example, the relevant command has already been configured.

- Queuing the commands to be sent to the device.

  As scripts are run against the device, the configuration they create is stored in the shared data area. When all the configuration to be sent has been accumulated, it is sent in a single communication session with the device.

- Prioritizing the order in which script functionality is executed.

  Each behavior script assigns a priority to the behavior object it creates. The controller script sorts behavior objects and executes them in their priority order.

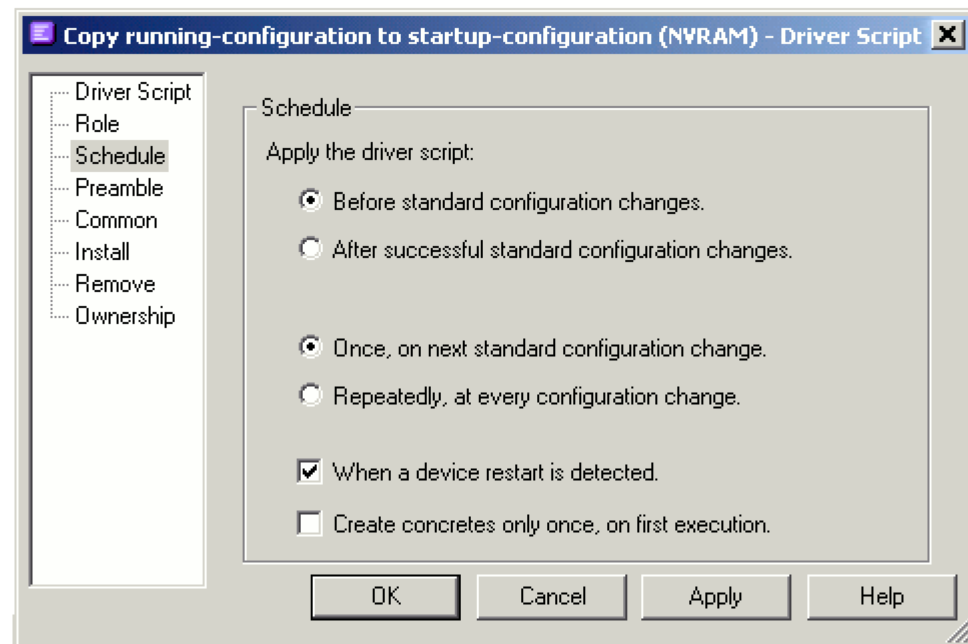
The following scripts feature in the example:

- Data scripts that define interface-specific configuration:
  - **setIpAddress.py** defines the IP address of an interface
  - **setDescription.py** defines the description of an interface
- A behavior script, **processInterfaces.py**, which creates a behavior object in the shared data area, indicating the priority in which the behavior must be run.
- The example illustrates a single behavior script, but there may be a number of behavior scripts set to run against a device. The processing performed by a behavior script may be dependent on some processing that has already been performed by another behavior script. For example, the class map name created by a behavior script may act as input to a subsequent behavior script that generates the policy map.
- A controller script, **BehaviorAndCommandController.py**, which sorts the stored behavior objects and calls an Execute method that is defined for every behavior.

Most of the processing that is carried out on existing and new configuration is performed by methods defined within the behavior script. However, these are not actually run until the controller script calls the behavior's Execute method.

The Oracle Communications IP Service Activator client provides a basic method for specifying the order in which scripts are applied to an object, enabling you to specify whether a script is run before or after the standard configuration changes made by the device driver. Figure 5–4 shows the Schedule page in the Driver Script dialog box where you can specify when the script is run.

***Figure 5–4 The Schedule Page***

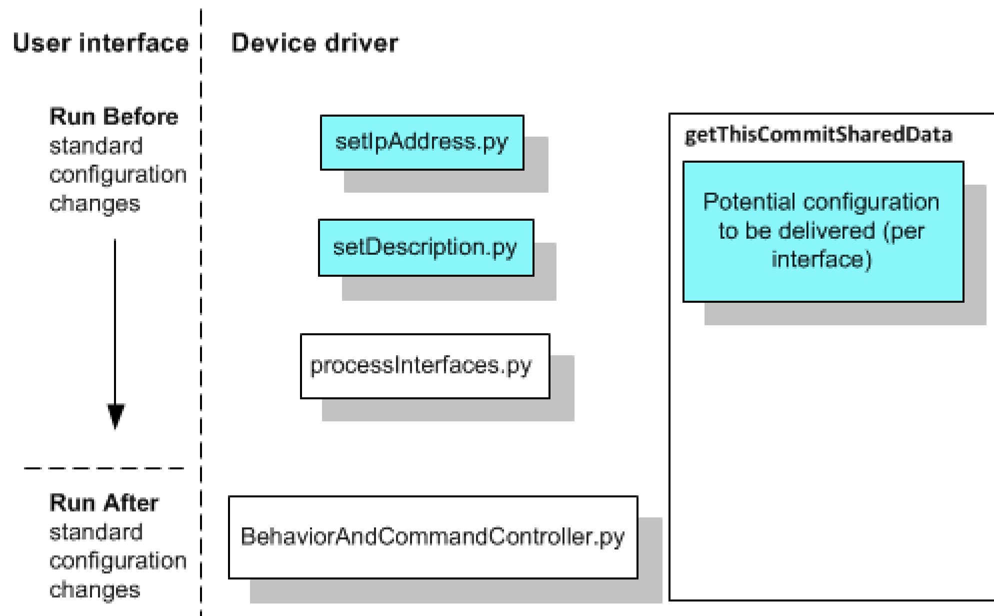



By specifying that data and behavior scripts run before standard configuration changes and the controller script runs after the changes, it is guaranteed that any data that the controller script needs to process will exist in the shared data area.

All of the scripts operate on the shared data area that is dedicated to data held for the current transaction only:

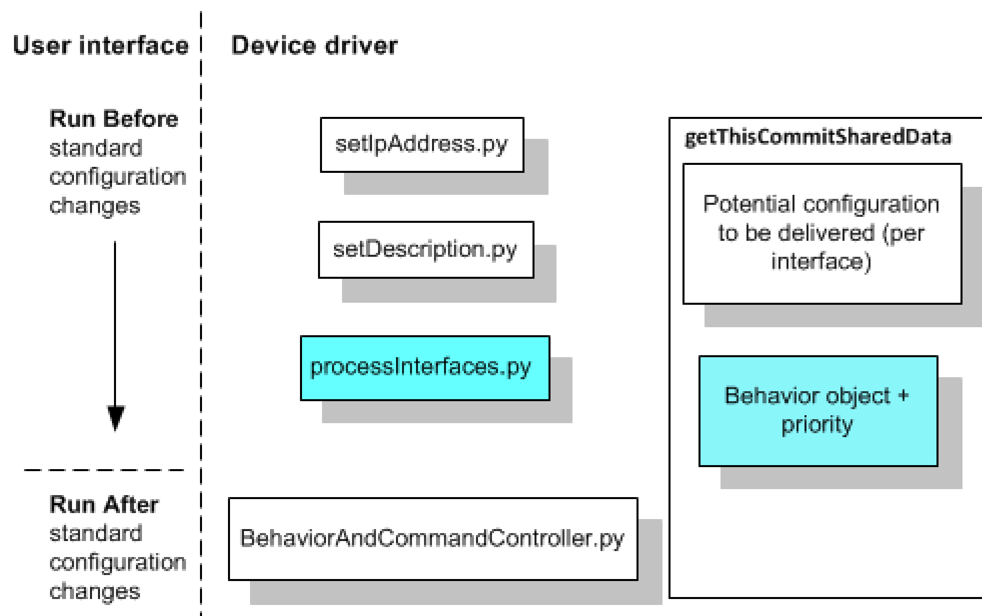
- The data scripts write information to the shared area, checking for the existence of stored potential configuration for the device, and adding their output to the storage area. Figure 5–5 illustrates the data script in operation.

***Figure 5–5 Data Scripts Writing Configuration Data to the Shared Data Area***




- The behavior script writes a behavior object to the area and stores information about the order in which it should be executed in relation to other behaviors (there may be multiple behavior scripts run against a device in a single transaction). Figure 5–6 illustrates the behavior script in operation.

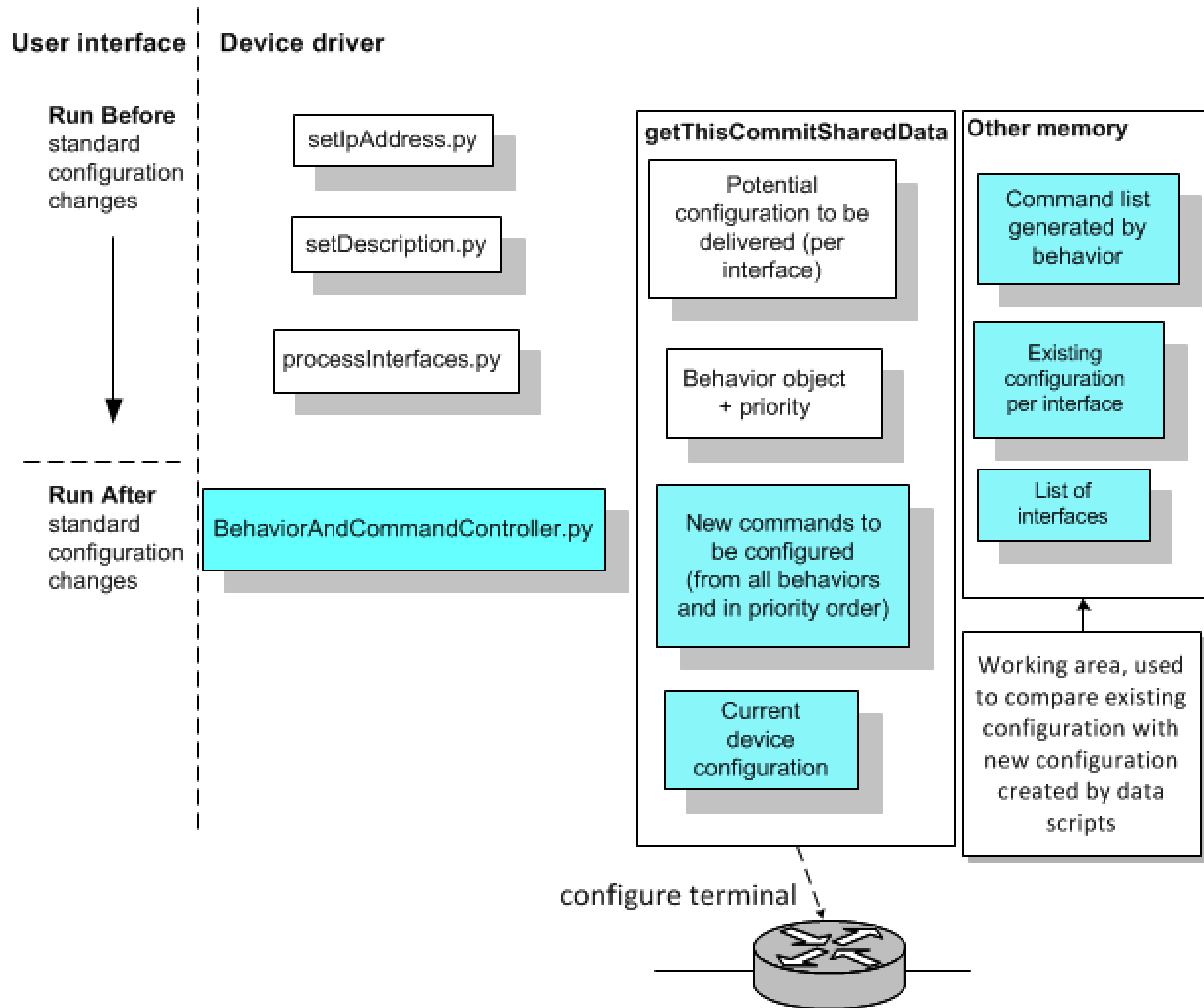
***Figure 5–6 Behavior Script Writing a Behavior Object to the Shared Data Area***




- The controller script processes any behavior objects written to the shared data area in their priority order, calling an Execute method defined for each behavior object.

This performs the body of the processing, retrieving and parsing configuration from the device (if it has not already been retrieved), comparing the existing configuration with the stored configuration created by the data scripts, and sending any configuration that needs to be sent to the device in a single communication session. Figure 5–7 illustrates the controller script in operation.

***Figure 5–7 Controller Script Processing Behavior Objects***




The following sections provide a listing of each script. The scripts update the getThisCommitSharedData area or read data from the area, storing data in a number of dictionaries. These are:

- **Interfaces**: created or updated by the data scripts with the IP address and description with which to configure an interface on a device, and by the behavior script to retrieve new commands for the device
- **Config**: used by the behavior script to store existing device configuration
- **Commands**: used by the behavior script to store the commands to be transmitted to the device, and by the controller script to retrieve commands for transmission to the device

- **Behavior**: used by the behavior script to store information about the priority in which it must be run in relation to other behavior scripts, and by the controller script to call each behavior's Execute method in the correct order.

The behavior script also creates additional dictionaries outside the getThisCommitSharedData area. These are used to process existing interface configuration so that it can be compared with potential new commands to be configured for an interface. These dictionaries are:

- **allinterfaces**: holds a complete list of the interfaces on the device
- **interfaceconfig**: holds the relevant commands (that is, commands that configure the interface's IP address and description) that are already configured on a device for a specific interface

The behavior script also creates a list of the commands that need to be sent to the device in a temporary area before adding this information to the commands dictionary in the **getThisCommitSharedData** area.

## Data Script: setIPAddress.py

The **setIPAddress.py** script defines an interface-specific IP address and mask and would be applied at interface level through the IP Service Activator client.

The sections that follow provide the **setIPAddress.py** script listing and an explanation of how the script works.

### Listing of setIPAddress.py

```
1   #Title: InterfaceIpAddressChanges.
    Version = '1.0'
    #IP Service Activator version: 7.0.0
    #Date: 1-Aug-2008




9   #begin preamble section
    # Copy this line and paste it into the context field, editing the
    # IP address and mask as required
    ip="0.0.0.0"
    mask="255.255.255.252"

15  #begin behavior section
    script_name = "INTERFACE_IPADDRESS"
    script_driver_type = "cisco"
    script_type = Interface
    script_device_role = Any
    script_interface_role = Any
    script_apply_when = Before
    script_repeat = False
    script_apply_on_restart = True

25  #begin common section
    # First make sure that there is a section representing interfaces
    if not _device.getThisCommitSharedData().has_key("interfaces"):
      _device.getThisCommitSharedData()["interfaces"]={}

30  interfaces = _device.getThisCommitSharedData()["interfaces"]
    if not interfaces.has_key(_interface.getInterfaceName() ):
      interfaces[_interface.getInterfaceName()]={}
```

```
34  #begin install section
    # Store the ip address command
36  ipCmd="ip address "+ ip + " " +mask
    interfaces[_interface.getInterfaceName()]["ip address"]=ipCmd

39  #begin remove section
    # Mark the ip address key as empty
    interfaces[_interface.getInterfaceName()]["ip address"]=""
```

### Explanation of setIPAddress.py

The script's processing starts by checking whether there is an interfaces key in the **getThisCommitSharedData** dictionary (line 27). If no key exists, the script creates one and makes the newly-created key a reference to a new dictionary. On lines 30-32, the script retrieves the content of the interfaces dictionary, and checks whether a key exists with the relevant interface name. If no key exists, the script creates one and makes the newly-created key a reference to a new dictionary. The interface IP address and mask is then stored at the correct point in the interfaces dictionary (lines 34-37).

## Data Script: setDescription.py

The **setDescription.py** script performs processing identical to that performed by **setIpAddress.py** except that an interface description replaces the IP address.

The section that follows provides the **setDescription.py** script listing.

### Listing of setDescription.py

```
1   #Title: InterfaceDescriptionChanges
    Version = '1.0'
    #IP Service Activator version: 7.0.0
    #Date: 31-Mar-2008




9   #begin preamble section
    description= "PE interface for customer XXX"

12  #begin behavior section
    script_name = "INTERFACE_DESCRIPTION"
    script_driver_type = "cisco"
    script_type = Interface
    script_device_role = Any
    script_interface_role = Any
    script_apply_when = Before
    script_repeat = False
    script_apply_on_restart = True

22  #begin common section
    # first make sure that there is a section representing interfaces
    if not _device.getThisCommitSharedData().has_key("interfaces"):
      _device.getThisCommitSharedData()["interfaces"]={}

    interfaces=_device.getThisCommitSharedData()["interfaces"]
    if not interfaces.has_key(_interface.getInterfaceName() ):
      interfaces[_interface.getInterfaceName()]={}

31  #begin install section
```

```
    # Set the description of the interface data
    descCmd="description "+ description
    interfaces[_interface.getInterfaceName()]["description"]=descCmd

36  #begin remove section
    # Mark the description key as empty
    interfaces[_interface.getInterfaceName()]["description"]=""
```

## Behavior Script: processInterfaces.py

The sections that follow provide the **processInterfaces.py** script listing and an explanation of how the script works.

### Listing of processInterfaces.py

```
1   #Title: InterfaceDescriptionChanges
    Version = '1.0'
    #IP Service Activator version: 7.0.0
    #Date: 31-Mar-2008




9   #begin preamble section
    #N/A

12  #begin behavior section
    script_name = "PROCESS_INTERFACES"
    script_driver_type = "cisco"
    script_type = Interface
    script_device_role = Any
    script_interface_role = Any
    script_apply_when = Before
    script_repeat = False
    script_apply_on_restart = True

22  #begin common section
    #Load the string module
24  import string
    class InterfaceProcessor:
26    def __init__( self ):
27      self.types=("description", "ip address" )
28      def getDeviceConfig(self, command ):
          if not _device.getThisCommitSharedData().has_key("config"):
            _device.getThisCommitSharedData()["config"]={}

34        # Log onto the device, send the command and stash the returned
          # config split into lines.
          if not _device.getThisCommitSharedData()["config"].has_key(command):
            _device.openSession()
            config = _device.deliverCommand(command )
            _device.closeSession()
            _
device.getThisCommitSharedData()["config"][command]=string.split(config,"\n")

40        # Return the content of the [config][command] key
          return _device.getThisCommitSharedData()["config"][command]

43        # Get configuration from the device
        def getInterfaceConfig( self ):
```

```
          config=self.getDeviceConfig("show running config")

47        allinterfaces={}
          interfaceconfig=None
49        for ind in range( 0 , len(config) ):
            str=string.lstrip( config[ind] )

            if ( str.startswith ("interface") ):
53            interfaceconfig={}
              allinterfaces[str]=interfaceconfig
            else:
            if( interfaceconfig ):
57            if( str == "!"):
                interfaceconfig=None
              else:
60              for type in self.types:
61                if ( str.startswith( type )  ):
62                  interfaceconfig[type]=str
          # else not in an interface and do need to look at the line.
          return allinterfaces

66      def Execute(self):

68        # Check for the commands dictionary
          if not _device.getThisCommitSharedData().has_key("commands"):
            _device.getThisCommitSharedData()["commands"]={}
71        # Check that priority 3 commands exist
          if not _device.getThisCommitSharedData()
["commands"].has_key(3):
            _device.getThisCommitSharedData()["commands"][3]=[]

76        # Check whether there is any required interface configuration
          if not _device.getThisCommitSharedData().has_key("interfaces"):
            return

80        # Retrieve the current configuration from the device
          allOldData=self.getInterfaceConfig()

83        # Loop through the interfaces with required data
          for (name, reqData) in _
device.getThisCommitSharedData()["interfaces"].items():
            commands=[]

87          # Get the existing interface config or create an empty object
            if( allOldData.has_key(name) ):
              oldData=allOldData[name]
            else:
              oldData={}
92            # Process each type of config the script can handle in turn
              if reqData.has_key(type):
94              if(  reqData[type] ):
                if( not oldData.has_key(type) or oldData[type] != reqData[type] ):
                  commands.append( reqData[type] )
                else:
                  if( oldData.has_key(type) ):
                    commands.append( "no " + oldData[type ] )

102        # If there are any interface commands, surround them by enter
          # and exit commands and add them to the list of commands to execute
          if( len( commands )):
```

```
            _device.getThisCommitSharedData()["commands"][3].append
("interface " + name)
            _device.getThisCommitSharedData()["commands"][3]=\
            _device.getThisCommitSharedData()["commands"][3] + commands

111         _device.getThisCommitSharedData()["commands"][3].
append("exit")

113 #begin install section
    if not _device.getThisCommitSharedData().has_key("behavior"):
      _device.getThisCommitSharedData()["behavior"]={}

    if not _device.getThisCommitSharedData()["behavior"].has_key(100):
      _device.getThisCommitSharedData()["behavior"][100]=[]

    _
device.getThisCommitSharedData()["behavior"][100].append(InterfaceProcessor())

#begin remove section
```

### Explanation of processInterfaces.py

The Common section (line 22) defines an **InterfaceProcessor** class that can perform the following tasks:

- Parse and store the configuration that is already on the device
- Check whether there is new configuration to be delivered to the device
- Compare new configuration with any existing configuration of the same type
- Store the commands to be delivered

The class can be divided into two parts. The first part defines methods for parsing, comparing and storing the configuration, the second part defines an Execute method that calls these methods. When the script is run, a behavior object is created in the shared data area, by a call to the **InterfaceProcessor** class. The **Execute** method is called and the script's processing is performed only when the controller script calls the method.

The imported string module (line 24) provides functionality required by the parsing process.

The **__init__** constructor method (line 26) is run automatically when the class name is called to create an instance. The method can be used to perform all the set up required for the class.

Line 27 creates a list of strings, where each string defines the start of a line of configuration that the script is designed to handle. This list of strings is assigned to the class variable types. This variable is used later in the script (line 93), and provides a more maintainable method of managing the data to be handled by the script. If the script needs to be updated to handle additional configuration in future, a new string can simply be added to the variable.

The **getDeviceConfig** method (line 28) is currently called with the show running config command. The method supports any other configuration command, however, and provides the type of functionality that you may wish to define in a Python module when using a class-based approach to the shared data area. The method checks whether the **getThisCommitSharedData** dictionary already has a copy of the running configuration (stored in the config dictionary). If not, it logs into the device, retrieves the config, splits it into lines and stores it in the config dictionary.

The **getInterfaceConfig** method calls **getDeviceConfig** with the show running config command. It then creates an **allinterfaces** dictionary (line 47) and an **interfaceconfig** variable and sets the variable to **None**. It then cycles through each line of configuration held in the **config** dictionary and checks for interface configuration (lines 49-50). Where interface configuration is found, the method creates an **interfaceconfig** dictionary (line 53) and adds a key labelled according to the name of the interface to the **allinterfaces** dictionary that points to the **interfaceconfig** dictionary. For each of the strings assigned to the class variable types (i.e. ip address and description), the method checks whether the line of config currently being processed starts with the string (line 61). If so, it creates a key in the **interfaceconfig** dictionary, taking the current value of **type** as the key name, and writes the current line of configuration to the key (line 62). When the end of interface configuration is reached (indicated by a `!` character) (line 57), the method clears the **interfaceconfig** variable to mark that the parser is outside of interface configuration. Finally, the method returns the content of the interfaces dictionary (line 64).

The **Execute** method is called by the controller script when it runs. It checks whether the commands dictionary exists within the **getThisCommitSharedData** dictionary and, if not, creates one (lines 69-70). It then checks that key 3 exists in the commands dictionary; this is the priority assigned to the commands generated by this particular behavior. There may be keys for commands generated by other behaviors in the commands dictionary. If the key does not exist, the method creates it (lines 72-74).

Before performing any further processing, the method checks whether there is any interface configuration waiting to be sent to the device (lines 77-79). The method checks whether an interfaces dictionary exists in the **getThisCommitSharedData** dictionary. This is the dictionary created or updated by any data scripts that have run against the device. If the dictionary does not exist, there is nothing to do and the method returns.

The method then retrieves the current configuration from the device by calling the **getInterfaceConfig** method (line 81). The return of **getInterfaceConfig** is the interfaces dictionary, so the **allOldData** variable is a pointer to a dictionary.

The method then loops through the interfaces for which configuration has been stored in the interfaces dictionary (lines 83-86), retrieving its name, value pairs (each pair consists of an interface name and the configuration to be put onto the device). The method processes each pair in turn, creating an area for the current interface's commands.

If the retrieved configuration (held in the interfaces dictionary) has a key whose name matches that currently being processed, the method assigns the content of the key (i.e. the interface name that points to the current interface configuration) to the variable **oldData**. If not, the method creates an empty dictionary (lines 87-91).

Then each type of configuration the script can handle is processed in turn and the method checks whether there is a command stored for the interface (line 93). If the command to be delivered is not an empty string, and if the command is not already configured on the device (line 94), or is configured and is not the same as the command currently stored for the interface (line 96), the method adds the command to the commands list for the interface (97). If the command to be sent is an empty string (that is, the configured command is no longer required), and the command is already configured on the device, the method adds the 'no' form of the command to the commands list for the interface (line 100).

If there are any commands in the commands list, surround them by enter and exit commands and add them to the **getThisCommitSharedData** dictionary's commands dictionary at the priority 3 key (lines 102-111). The commands dictionary stores the commands to execute and their order of execution.

The Install section (line 113) is the only section of the script that will be executed when the script is run against the device. It writes a behavior object to the **getThisCommitSharedData** dictionary by calling the **InterfaceProcessor** method (line 119). Behavior objects are stored according to their priority within a behavior dictionary.

## Controller Script: BehaviorAndCommandController.py

The sections that follow provide the **BehaviorAndCommandController.py** script listing and an explanation of how the script works.

### Listing of BehaviorAndCommandController.py

```
1  #Title: CommandSender
   Version = '1.0'
   #IP Service Activator version: 7.0.0
   #Date: 1-Mar-2008



9  #begin preamble section
   #N/A
12 #begin behavior section
   script_name = "Command Sender"
   script_driver_type = "cisco"
   script_type = Device
   script_device_role = Any
   script_interface_role = Any
   script_apply_when = After
   script_repeat = true
   script_apply_on_restart = True

22 #begin common section

   #begin install section
25 if ( _device.getThisCommitSharedData().has_key("behavior") ):
     behaviors = _device.getThisCommitSharedData()["behavior"]
     ordering  = behaviors.keys()
     ordering.sort()
     for order in ordering:
       for behavior in behaviors[ order ] :
31       behavior.Execute()

33 if _device.getThisCommitSharedData().has_key("commands") :
     _device.openSession()
     _device.deliverCommand("configure terminal")

37   commands = _device.getThisCommitSharedData()["commands"]
     ordering = commands.keys()
     ordering.sort()
     for order in ordering:
       for command in commands[ order ] :
         _device.deliverCommand(command)

44   _device.closeSession()
```

### Explanation of BehaviorAndCommandController.py

The script first checks for the behavior dictionary in the **getThisCommitSharedData** dictionary (line 25). If the dictionary exists, the script retrieves its keys (each behavior

script has written a key to the behavior dictionary to indicate the priority in which it should be executed), sorts the keys/priorities and calls each behavior object's **Execute** method in turn (lines 26-31).

After processing the behavior object, the script checks whether this has resulted in commands being written to the **getThisCommitSharedData** dictionary's commands dictionary. The script sorts the stored commands by key, loops through all of the stored commands and sends them to the device (lines 37-42).

## Error Reporting in Behavior Scripts

If you follow the guidelines outlined for using the shared data area, a Behavior script's functionality is not executed until it is called by the Controller script. This means that the **_result** variable that indicates the success or failure of the script that is currently running cannot be used to report on the Behavior script. This is because the script that is currently running when the Behavior script's functionality is executed is the Controller script.

To report on errors that occur when the behavior object is executed, the Behavior script's ID and failure information must be stored within the behavior object created by the script. These details can then be retrieved by the Controller script when it runs.

The Behavior script's class constructor method should define variables to hold the values of the **_id** and **_failure_code** variables, as follows:

```
class InterfaceProcessor
  def __init__ (self):
    self.id=_id
    self.failure=_failure_code
```

The script's **Execute** method checks for a failure condition and, if a failure occurred, returns the variable values and a message about the failure, as follows:

```
  if (failure):
    return (self.id,self.failure,'behavior script InterfaceProcessor failed')
```

The controller script uses the **_result** variable's **sendScriptObjectFailure** method to retrieve the behavior script's failure details, as follows:

```
result=behavior.Execute()
  if (result):
  (id,failure,message)=result
  _result.sendScriptObjectFailure(id,failure,message)
```

These details are sent to the policy server and the message displayed in the current faults pane in the user interface.

# 6

# Monitoring and Troubleshooting Scripts

This chapter explains the features available for checking scripts.

## Checking the Status of Scripts

Once you have applied driver scripts you can check the points in the system at which they will run.

Device scripts can be applied to different levels of objects in the system: customers, VPNs, sites, networks, devices, interfaces, sub-interfaces or VC endpoints. You can check each object to see which scripts currently apply to it.

To check the status of a script:

1. In the Oracle Communications IP Service Activator client, double-click the relevant object (for example, a site, device or interface) from the hierarchy tree or the topology map.

   The configuration applied to this object appears in the Details pane.

2. Click the **Driver Scripts** tab to view relevant scripts.

Just as with rules, each abstract (parent) script is followed by a list of the concrete scripts that have been created at appropriate interfaces. Parent scripts appear on a white background if they have been set up on the selected object and on a gray background if they have been inherited from a higher-level object.

Figure 6–1 shows the Driver Scripts tab on the Details pane.

***Figure 6–1 The Driver Scripts Tab***

Details - Driver Scripts - VPN_PE3

| Name | State | Level | Driver Type | Applies To | Device Role | Interface Role |
|---|---|---|---|---|---|---|
| Netflow Activation | | VPN_PE3 | cisco | Interface | Any Role | Any Role |
| SAA DNS Connect | | VPN_PE3 | cisco | Interface | Any Role | Any Role |
| SAA HTTP Web Server ... | | VPN_PE3 | cisco | Interface | Any Role | Any Role |
| Save to NVRAM | | VPN_PE3 | cisco | Device | Any Role | Any Role |

Rules are listed under the following headings:

- **Name**: the name of the script.
- **State**: the current status of the script. For concrete scripts the status is one of the following:

  - **Inactive**: the script has been created, but not yet propagated to the devices.
  - **Active**: the script has been propagated to proxy agents, but is not yet configured on a device.
  - **Installed**: the script has been propagated to proxy agents and has been successfully installed on the designated device.
  - **Failed**: the proxy agent experienced a failure trying to install the script and it has therefore been discarded. Note that for a repeating script a further attempt will be made to install the script on the next propagate.
  - **Rejected**: the script has been rejected by the device driver (for example because of a syntax error).
- **Level**: the level at which the script is implemented.
- **Driver Type**: the device driver that will run the script.
- **Script Type**: the type of script. Values are: **Device**, **Interface**, **Sub-interface**, **ATM PVC**, or **Frame Relay PVC**.
- **Device Role**: the role of devices that this script will be applied to. Values are: **Access**, **Gateway**, **Core**, **Shadow**, or **Any Role**.
- **Interface Role**: for Interface, Sub-interface, ATM PVC and FR PVC scripts, the interface role that this script will be applied to. Values are **Core**, **Local**, **Access**, or **Any Role**.
- **Installed**: indicates when this script will be applied relative to other configuration changes applied as part of the propagate. Value is either **Before config changes** or **Following successful config changes**.
- **Frequency**: indicates how often the configuration will be applied. Value is either **Once only** (that is, on the next propagate) or **Repeat** (on each propagate until the script is removed).
- **Install on restart:** indicates if the script is to be run when a device restart is detected. Value is either **True** or **False**.
- **Owner**: if ownership of the script has been specified the value is the owner's username.
- **Owner Group**: the group to which the owner belongs.

## Understanding Warnings and Error Messages

Table 6–1 lists the warnings and errors that may be raised if scripts are incorrectly applied.

---

**Note:** Scripts are not checked or validated by the IP Service Activator client.

---

*Table 6–1 Script Warning and Error Messages*

| Number | Severity | String | Explanation |
|---|---|---|---|
| 2100 | Warning | `Driver Script` *`name`* `will not have any effect when applied here.` | Raised if a script is applied at too low a level for its type, for example, a device script is linked to an interface. |
| 2101 | Warning | `Driver Script` *`name`* `will not have any effect when applied here because of the type of the script.` | Raised if a script is applied at an object where it can't take effect because of the roles specified, for example, an Access interface script is linked to a Local interface. |
| 2205 | Error | `Driver Script has no effect due to mismatch with Device Driver type.` | Raised if a script cannot be applied to the specified object because it is not applicable to the driver managing the site. |
| 3309 | Error | `Driver script failed with error details:` *`details`* | Raised if a driver script fails when run by the device driver. Any user information incorporated within the driver script is displayed in the *details* string. |

# Checking Logs

All device configuration changes applied by the Cisco device driver are recorded in a specific log file. You can check these log files to see if configuration applied by driver scripts is being successfully applied to a device.

On Solaris systems, the log files are in **/opt/OracleCommunications/IPServiceActivator/AuditTrails**

On Windows systems, the log files are created in P**rogram Files\ Oracle Communications\IP Service Activator\AuditTrails**

Commands delivered to the devices are also logged in the Device Configuration Log.

To check the Device Configuration Log

1. On the System tab, open the **System Logs** folder.
2. Choose **Device Configuration Log**.

   The Device Configuration Log appears in the details pane.

# 7 Definition of Standard Methods

This chapter describes the standard methods that scripts can use. Many of these methods are dependent on the context in which they are called.

## Summary of Methods

Table 7–1 list the methods that scripts use.

*Table 7–1 Standard Methods for Scripts*

| Method | Context | Purpose |
|---|---|---|
| setCode | General | Sets the return status of a script run. |
| setDetails | General | Sets a result string to be passed back to the user, typically explaining an error. |
| sendScriptObjectiveFailure | General | Creates a notification message indicating script failure to be passed back to the policy server. |
| openSession | Device | Connects to the device, authenticates and gets to the enable prompt. This method *must* be called before any call to **deliverCommand**. |
| deliverCommand | Device | Sends a command to the device followed by a new line. **openSession** *must* have been called before this method. |
| closeSession | Device | Closes the connection to the device. This method should be called after all commands have been delivered. |
| getIpAddress | Device | Gets the primary IP address of the device, that is, the IP address that the driver uses to contact the device. |
| getIos | Device | Gets the Cisco IOS version.<br>This method is relevant to the Cisco IOS device driver only. |
| getOs | Device | Gets the operating system version. |
| getDeviceType | Device | Gets the device type.<br>This method is relevant to the Cisco IOS device driver only. |
| getFeatureSet | Device | Gets the feature set.<br>This method is relevant to the Cisco IOS device driver only. |
| getNumberOfInterfaces | Device | Gets the number of interfaces on the device. This is the number of managed interfaces, not the total number of interfaces. |

*Table 7–1 (Cont.) Standard Methods for Scripts*

| Method | Context | Purpose |
|---|---|---|
| getInterfaces | Device | Gets the interface object at *index* position in the total set of interfaces. The total number of interfaces is given by the call. Valid *index* values are from 0 to `getNumberofInterfaces() - 1` |
| log | Device | Logs a message string to the driver log. |
| getThisCommitSharedData | Device | Gets the information stored in the shared data area for the current transaction only. Information is stored as a dictionary or class, depending on whether the -SharedDataModule driver command-line option is used. |
| getLifetimeSharedData | Device | Gets the information that is stored in the shared data area for the lifetime of the device (since the device was last managed or the device driver restarted). Information is stored as a dictionary or class, depending on whether the -SharedDataModule driver commadn-line option is used. |
| getInterfaceName | Interface | Gets the name fo the interface. |
| getIpAddress | Interface | Gets the IP address of the interface. |
| getVipType | Interface | Gets type of the VIP the interface is on, if known.<br>This method is relevant to the Cisco IOS device driver only. |
| getAdapterType | Interface | Gets the type of physical card the interface is on.<br>This method is relevant to the Cisco IOS device driver only. |
| getNumberOfFramePvcs | Interface | Gets the number of Frame Relay PVCs on the interface. This is the number of managed PVCs, not the total number of PVCs. |
| getNumberOfAtmPvcs | Interface | Gets the number of ATM PVCs on the interface. This is the number of managed PVCs, not the total number of PVCs. |
| getFramePvc | Interface | Gets a Frame Relay PVC object. |
| getAtmPvc | Interface | Gets an ATM PVC object. |
| getVpi | ATM PVC | Gets the PVC VPI. |
| getVci | ATM PVC | Gets the PVC VCI. |
| getDlci | FR PVC | Gets the PVC DCLI. |

# General Context

The General context is available to all scripts. This context provides the **_result** object.

## The _result Object

The **_result** object passes a result out of a script. The type of this object is ScriptResultPtr.

Table 7–2 lists the variables of the **_result** object.

*Table 7–2 The _result Object Variables*

| Variable | Purpose |
|---|---|
| _id | Indicates the unique ID of the concrete script. |

*Table 7–2 (Cont.) The _result Object Variables*

| Variable | Purpose |
|---|---|
| _failure_code | Indicates the code a particular script should use if the script does not run successfully. The value of the variable differes depending on the script's scheduling properties and whether it is set to run on remove. |

**Note:** You cannot assign values to these variables. If these values need to be changed, you must call the **sendScriptObjectFailure** method on the **_result** object.

The following methods can be called on the **_result** object:

- The setCode Method
- The setDetails Method
- The sendScriptObjectFailure Method

## The setCode Method

The **setCode** method sets the status of a script run.

Table 7–3 shows the arguments for the **setCode** method.

*Table 7–3 setCode Method Arguments*

| Argument | Description | Default |
|---|---|---|
| code | OK or FAILED | OK |

### Syntax

The syntax for the **setCode** method is:

```
setCode ({OK | FAILED})
```

### Exceptions

N/A

### Return

N/A

## The setDetails Method

The **setDetails** method sets a result string to be passed back to the user.

Table 7–4 shows the arguments for the **setDetails** method.

*Table 7–4 setDetails Method Arguments*

| Argument | Description | Default |
|---|---|---|
| *details* | A string that can be used to pass a reason for failure or the result of a successful script. | " " |

### Syntax

The syntax for the **setDetails** method is:

```
setDetails (details)
```

#### Exceptions

N/A

#### Return

N/A

### The sendScriptObjectFailure Method

The **sendScriptObjectFailure** method creates a notification message indicating script failure to be passed back to the policy server.

Table 7–5 shows the arguments for the **sendScriptObjectFailure** method.

***Table 7–5 sendScriptObjectFailure Method Arguments***

| Argument | Description |
|---|---|
| *id* | A variable that indicates the script's ID number held by the IP Service Activator _id variable. |
| *failure_code* | A variable that indicates the script's failure code held by the IP Service Activator _failure_code variable. |
| *message* | A string that indicates the cause of the failure. The *message* is passed back to the policy server and displayed in the Current Faults pane of the IP Service Activator client. |

#### Syntax

The syntax for the **sendScriptObjectFailure** method is:

```
sendScriptObjectFailure (id,failure_code,'message'
```

#### Exceptions

N/A

#### Return

N/A

# Device Context

The Device context is available to Device, Interface, Sub-interface, ATM PVC and Frame PVC scripts. This context provides the **_device** object.

## The _device Object

The **_device** object provides access to the device functions. The type of this object is PythonCiscoProxyDevicePtr.

The following methods can be called on the **_device** object:

- The openSession Method
- The deliverCommand Method
- The closeSession Method
- The getIpAddress Method

- The getIos Method
- The getOs Method
- The getDeviceType Method
- The getFeatureSet Method
- The getNumberOfInterfaces Method
- The getThisCommitSharedData Method
- The getLifetimeSharedData Method
- The getInterface Method
- The log Method
- The auditLog Method


## The openSession Method

The **openSession** method connects to the device, authenticates and gets to the Enable prompt. This method *must* be called before any call to **deliverCommand**.

The **openSession** method has no arguments.

### Syntax

The syntax for the openSession method is:

```
openSession( )
```

### Exceptions

RuntimeError will be raised if a session cannot be established.

### Return

N/A

## The deliverCommand Method

The **deliverCommand** method sends a command to the device followed by a CRLF. The **openSession** method *must* have been run before this method.

Table 7–6 shows the arguments for the **deliverCommand** method.

***Table 7–6 deliverCommand Method Arguments***

| Argument | Description |
|---|---|
| *command* | String command to send to the device. |

### Syntax

The syntax for the **deliverCommand** method is:

```
[result=]deliverCommand('command')
```

### Exceptions

RuntimeError will be raised if the command cannot be delivered.

### Return

The **deliverCommand** method returns a string that is the result of executing the command, for example, the output of a "show version".

## The closeSession Method

The **closeSession** method closes the connection to the device. This method should be called after all commands have been delivered. If it is not called, the connection will be automatically closed.

The **closeSession** method has no arguments.

### Syntax

The syntax for the **closeSession** method is:

```
closeSession( )
```

### Exceptions

RuntimeError will be raised if a session cannot be closed or was never opened.

### Return

N/A

## The getIpAddress Method

The **getIpAddress** method gets the primary IP address of teh device, that is, the IP address the driver uses to contact the device.

The **getIpAddress** method has no arguments.

### Syntax

The syntax for the **getIpAddress** method is:

```
getIpAddress( )
```

### Exceptions

N/A

### Return

The **getIpAddress** method returns an integer that represents the IP Address in host byte order.

## The getIos Method

The **getIos** method gets the IOS version of the device. This method is relevant to the Cisco IOS driver only.

The **getIos** method has no arguments.

### Syntax

The syntax for the **getIos** method is:

```
getIos( )
```

### Exceptions

N/A

#### Return

The **getIos** method returns a string that represents the IOS version, for example, 11.1(12XP1) or 12.1(1)E.

### The getOs Method

The **getOs** method gets the operating system of the device. This method is relevant to all devices.

The **getOs** method has no arguments.

#### Syntax

The syntax for the **getOs** method is:

```
getOs( )
```

#### Exceptions

N/A

#### Return

The **getIos** method returns a string that represents the OS version.

### The getDeviceType Method

The **getDeviceType** method gets the device type. This method is relevant to the Cisco IOS driver only.

The information is obtained by querying the driver by a **show version** command.

The **getDeviceType** method has no arguments.

#### Syntax

The syntax for the **getDeviceType** method is:

```
getDeviceType( )
```

#### Exceptions

N/A

#### Return

The **getDeviceType** method returns a string that represents the device type, for example, RSP or C7200.

### The getFeatureSet Method

The **getIos** method gets the character(s) representing the IOS feature set available on the device. This method is relevant to the Cisco IOS driver only.

The **getFeatureSet** method has no arguments.

#### Syntax

The syntax for the **getFeatureSet** method is:

```
getFeatureSet( )
```

#### Exceptions

N/A

#### Return

The **getFeatureSet** method returns a string of alphanumeric characters that represents the IOS feature set, for example, `p` for Service Provider feature set, `i` for IP subset. Check Cisco documentation for the complete list.

### The getNumberOfInterfaces Method

The **getNumberOfInterfaces** method gets the number of interfaces on the device. This is the number of managed interfaces, not the total number of interfaces.

The **getNumberOfInterfaces** method has no arguments.

#### Syntax

The syntax for the **getNumberOfInterfaces** method is:

```
getNumberOfInterfaces( )
```

#### Exceptions

N/A

#### Return

The **getIos** method returns an integer that represents the number of interfaces on the device.

### The getThisCommitSharedData Method

The **getThisCommitSharedData** method gets the information stored in the shared data area for the current transaction only.

The **getThisCommitSharedData** method has no arguments.

#### Syntax

The syntax for the **getThisCommitSharedData** method is:

```
[result=]getThisCommitSharedData( )
```

#### Exceptions

Not applicable.

#### Return

The **getThisCommitSharedData** method returns a string that represents the information stored in the data area for the current transaction.

### The getLifetimeSharedData Method

The **getLifetimeSharedData** method gets the information stored in the shared data area for the lifetime of the device; that is, since the device was last managed or the device driver was restarted.

The **getLifetimeSharedData** method has no arguments.

#### Syntax

The syntax for the **getLifetimeSharedData** method is:

```
[result=]getLifetimeSharedData( )
```

#### Exceptions

N/A

#### Return

The **getLifetimeSharedData** method returns a string that represents the information stored in the shared data of the area since the device was last managed or teh device driver was restarted.

### The getInterface Method

The **getInterface** method gets the interface object at **index** position in the total set of interfaces. The total number of interfaces is given by the **getNumberOfInterfaces** method. Valid index values are from 0 to the number given by **getNumberOfInterfaces** minus 1.

Table 7–7 lists the arguments for the **getInterface** method.

*Table 7–7 getInterface Method Arguments*

| Argument | Decription |
|---|---|
| *index* | Interface to fetch. |

#### Syntax

The syntax for the **getInterface** method is:

```
getInterface(index)
```

#### Exceptions

RuntimeError will be raised if *index* is not valid.

#### Return

The **getInterface** method returns the **PythonCiscoProxyInterfacePtr** object if *index* is valid, **None** if *index* is not valid.

### The log Method

The **log** method logs a specified string to the driver log.

Table 7–8 lists the arguments for the **log** method.

*Table 7–8 log Method Arguments*

| Argument | Decription |
|---|---|
| *string* | Message to write to log. |

#### Syntax

The syntax for the **log** method is:

```
log(string)
```

#### Exceptions

N/A

#### Return

N/A

### The auditLog Method

The **auditLog** method logs a message to the device driver's audit trail log.

Table 7–9 lists the arguments for the **log** method.

*Table 7–9 auditLog Method Arguments*

| Argument | Decription |
|---|---|
| *message* | Message to write to log. |

#### Syntax

The syntax for the **auditLog** method is:

```
auditLog('message')
```

#### Exceptions

N/A

#### Return

N/A

# Interface Context

The Interface context is available to Interface, Sub-interface, ATM PVC and Frame PVC scripts. This context provides the **_interface** object.

## The _interface Object

The **_interface** object provides access to interface function. The type of this object is PythonCiscoProxyInterfacePtr.

The following methods can be called on the **_interface** object:

- The getInterfaceName Method
- The getIpAddress Method
- The getVipType Method
- The getAdapterType Method
- The getNumberOfFramePvcs Method
- The getNumberOfAtmPvcs Method
- The getFramePvc Method
- The getAtmPvc Method

### The getInterfaceName Method

The **getInterfaceName** method gets the name of the interface.

The **getInterfaceName** method has no arguments.

#### Syntax

The syntax for the **getInterfaceName** method is:

```
getInterfaceName( )
```

#### Exceptions

RuntimeError will be raised if **index** is not valid.

#### Return

The **getInterfaceName** method returns a string that represents the interface name, for example, Serial1/0 or Ethernet1.

### The getIpAddress Method

The **getIpAddress** method gets the IP address of the interface.

The **getIpAddress** method has no arguments.

#### Syntax

The syntax for the **getIpAddress** method is:

```
getIpAddress( )
```

#### Exceptions

N/A

#### Return

The **getIpAddress** method returns an integer that represents the IP address in host byte order.

### The getVipType Method

The **getVipType** method gets the VIP adapter (if any). This method is relevant to the Cisco IOS driver only.

The **getVipType** method has no arguments.

#### Syntax

The syntax for the **getVipType** method is:

```
getVipType( )
```

#### Exceptions

N/A

#### Return

The **getVipType** method returns a string that represents the VIP. Conversion is as follows:

- VIP = 'VIP'
- VIP2-10 = 'VIP2_10'
- VIP2-15 = 'VIP2_15'
- VIP2-20 = 'VIP2_20'
- VIP2-40 = 'VIP2_40'

- VIP2-50 = 'VIP2_50'
- VIP4-80 = 'VIP4_80'

If the interface is not on a VIP or the type is unknown, an emtpy string is returned.

### The getAdapterType Method

The **getAdapterType** method gets the type of physical card that the interface is on. This method is relevant to the Cisco IOS driver only.

The **getAdapterType** method has no arguments.

#### Syntax

The syntax for the **getAdapterType** method is:

```
getAdapterType( )
```

#### Exceptions

N/A

#### Return

The **getIpAddress** method returns a string that represents the card type.

### The getNumberOfFramePvcs Method

The **getNumberOfFramePvs** method gets the number of Frame Relay PVCs on the interface. This is the number of managed PVCs, not the total number of PVCs.

The **getNumberOfFramePvcs** method has no arguments.

#### Syntax

The syntax for the **getNumberOfFramePvcs** method is:

```
getNumberOfFramePvcs( )
```

#### Exceptions

N/A

#### Return

The **getNumberOfFramePvcs** method returns an integer that represents the number of Frame Relay PVCs.

### The getNumberOfAtmPvcs Method

The **getNumberOfAtmPvcs** method gets the number of ATM PVCs on the interface. This is the number of managed PVCs, not the total number of PVCs.

The **getNumberOfAtmPvcs** method has no arguments.

#### Syntax

The syntax for the **getNumberOfAtmPvcs** method is:

```
getNumberOfAtmPvcs( )
```

#### Exceptions

N/A

#### Return

The **getNumberOfAtmPvcs** method returns an integer that represents the number of ATM PVCs.

### The getFramePvc Method

The **getFramePvc** method gets the Frame Relay PVC object at **index** position in the total set of Frame PVCs. The total number of Frame PVCs is given by the **getNumberOfFramePvcs** method. Valid index values are from 0 to the return of **getNumberOfFramePvcs** minus 1.

Table 7–10 shows the arguments for the **getFramePvc** method.

***Table 7–10 getFramePvc Method Arguments***

| Argument | Description |
|---|---|
| *index* | Frame Relay PVC to fetch. |

#### Syntax

The syntax for the **getFramePvc** method is:

```
getFramePvc(index)
```

#### Exceptions

RuntimeError will be raised if **index** is not valid.

#### Return

The **getFramePvc** method returns the **PythonCiscoProxyFramePtr** object if *index* is valid, **None** if *index* is not valid.

### The getAtmPvc Method

The **getAtmPvc** method gets the ATM PVC object at **index** position in the total set of ATM PVCs. The total number of ATM PVCs is given by the **getNumberOfAtmPvcs** method. Valid index values are from 0 to the return of **getNumberOfAtmPvcs** minus 1.

Table 7–11 shows the arguments for the **getAtmPvc** method.

***Table 7–11 getAtmPvc Method Arguments***

| Argument | Description |
|---|---|
| *index* | ATM PVC to fetch. |

#### Syntax

The syntax for the **getAtmPvc** method is:

```
getAtmPvc(index)
```

#### Exceptions

RuntimeError will be raised if **index** is not valid.

#### Return

The **getAtmPvc** method returns the **PythonCiscoProxyFramePtr** object if *index* is valid, **None** if *index* is not valid.

# ATM PVC Context

The ATM PVC context is available to all ATM PVC scripts. This context provides the _ **atm_pvc** object.

## The _atm_pvc Object

The **_atm_pvc** object provides access to ATM PVC scripts. The type of this object is PythonCiscoProxyAtmPvcPtr.

The following methods can be called on the **_atm_pvc** object:

- The getVpi Method
- The getVci Method

### The getVpi Method

The **getVpi** method gets the PVC VPI.

The **getVpi** method has no arguments.

#### Syntax

The syntax for the **getVpi** method is:

```
getVpi( )
```

#### Exceptions

N/A

#### Return

The **getVPI** method returns an integer that represents the VPI of the PVC.

### The getVci Method

The **getVci** method gets the PVC VCI.

The **getVci** method has no arguments.

#### Syntax

The syntax for the **getVci** method is:

```
getVci( )
```

#### Exceptions

N/A

#### Return

The **getVci** method returns an integer that represents the VCI of the PVC.

# Frame PVC Context

The Frame PVC context is available to all Frame Relay PVC scripts. This context provides the **_frame_pvc** object.

## The _frame_pvc Object

The **_frame_pvc** object provides access to Frame Relay PVC functions. The type of this object is PythonCiscoProxyFramePvcPtr.

The following methods can be called on the _frame_pvc object:

- The getDlci Method

### The getDlci Method

The **getDlci** method gets the PVC Data Link Connection Identifier (DLCI).

The **getDlci** method has no arguments.

#### Syntax

The syntax for the **getDlci** method is:

```
getDlci( )
```

#### Exceptions

N/A

#### Return

The **getDlci** method returns an integer that represents the DLCI of the PVC.

# A

# Pre-defined Scripts

This appendix describes the pre-defined scripts that are supplied with Oracle Communications IP Service Activator.

## Save to NVRAM

The **NVRAM.py** script copies the Cisco running configuration to startup configuration (NVRAM). This is particularly useful if a number of changes have taken place during the current run period of the system and ensures these changes are saved for any future restarts.

### Device Driver

Cisco IOS device driver

### Objects Applied To

Script Type = Device

Device Role = Any Role

### Scheduling Requirements

Installed = After configuration changes

Frequency = Once only

Install on restart = False

### Variables Required

N/A

### Commands Applied During Install

```
copy running-config startup-config
```

### Commands Applied During Remove

N/A

## Add VLAN to CatOS

The **CatosAddVlna.py** script adds a VLAN to a CatOS switch.

### Device Driver

CatOS script driver

### Objects Applied To

Script Type = Device

Device Role = Any Role

### Scheduling Requirements

Installed = After configuration changes

Frequency = Repeatedly

Install on restart = False

### Variables Required

newVlanNum

newVlanName

### Commands Applied During Install

```
copy running-config startup-config
```

### Commands Applied During Remove

N/A

## Force a FastStart Mode Exit for Cisco Devices

The **NullCiscoScript.py** script forces Cisco devices to exit from the FastStart mode.

### Device Driver

Cisco IOS device driver

### Objects Applied To

Script Type = Device

Device Role = Any Role

### Scheduling Requirements

Installed = After configuration changes

Frequency = Once only

Install on restart = False

### Variables Required

N/A

### Commands Applied During Install

N/A

### Commands Applied During Remove

N/A

B

# Sample Scripts for Using the Shared Data Area

This appendix provides sample scripts. These scripts follow the logic outlined in "An Example of Using the Shared Data Area".

## Sample Python Module

The following script is supplied as a sample only and is not intended for deployment. Oracle Support does not provide support for this script.

When applied to the device driver using the command-line parameter **-SharedDataModule**, the following script defines the structure of the shared data area.

```
# This shared info module does not handle removing behavior objects
# from lifetime shared memory. One way to deal with this issue is for
# each behavior object to define an idea of identity and the shared
# Module to insert behaviors in a structure that can be accessed by
# identity so that specific objects can be removed.

# Python "reflection" would be needed to help solve the problem in this
# implementation.
class SharedInfo:
  # In these functions get and set are methods to get data
  # from/insert data into the shared area functions with Add relate
  # to data to be treated as an add action(normally added by the
  # "on install" part of a data script functions, with Remove
  # relate to data to be treated as a remove action (normally added
  # by the "on remove" part of a data script

  def __init__(self):
    self.behaviors={}         # Dictionary for behavior objects.
    self.data={"add":{},"remove":{}}  # Dictionary for data.- broken into add and
remove sections.
    self.addInterface={}              # Dictionary for add data to interfaces.
    self.removeInterface={}  # Dictionary for remove data to interfaces.
    self.general={}

  # Some simple functions to ensure that objects exist before being
  # accessed.
  def fetchDict(self, container, key):
    if( not container.has_key(key) ): container[key]={}
    return container[key]
  def fetch(self, container, key):
    if( not container.has_key(key) ): container[key]=[]
    return container[key]
  def insert(self, container, key, value ):
    if( not container.has_key(key) ): container[key]=[]
```

```
    container[key].append(value)

  # Insert the behavior object into the behaviors dictionary, keying
  # on the priority of the behavior object.
  def insertBehavior(self, behavior):
        self.insert( self.behaviors, behavior.priority(), behavior )
  # Return all of the registered behavior objects
  def fetchBehavior(self):
    return self.behaviors

  #Insert data keyed by data type and interface name.
  def setInterfaceAddData(self, iface,  name, data):
        self.insert(  self.fetchDict( self.addInterface, name), iface, data)
  def setInterfaceRemoveData(self, iface, name, data):
        self.insert(  self.fetchDict( self.removeInterface, name), iface, data)


  # Get data of type "name"  for all interfaces
  def getInterfaceAddData(self,  name):
        return  self.fetchDict( self.addInterface, name)
  def getInterfaceRemoveData(self, iface, name):
        return  self.fetchDict( self.removeInterface, name)
  # Get data of type "name" for this interface only.
  def getThisInterfaceAddData(self, iface, name):
        return self.fetch( self.fetchDict( self.addInterface, name), iface)
  def getThisInterfaceRemoveData(self, iface, name):
        return self.fetch( self.fetchDict( self.removeInterface, name), iface)

# Data for a given key is stored as a list ( in case multiple
# scripts register against the same key). "TrustMe" functions can
# be used when the calling script is sure that there will
# be an entry for the key, and that there is only one entry,
# or the first entry is the only one it cares about.
def getInterfaceAddDataTrustMe(self, iface, name):
        return self.fetch( self.fetchDict( self.addInterface, name), iface)[0]
  def getInterfaceRemoveDataTrustme(self, iface, name):
        return self.fetch( self.fetchDict( self.removeInterface, name), iface)[0]

  # Generic get and set data methods for add and remove
  def setAddData(self, name, data):
        self.insert(  self.data["add"], name, data)
  def setRemoveData(self, name, data):
        self.insert(  self.data["remove"], name, data)
  def getAddData(self, name):
        return self.fetch( self.data["add"], name)
  def getRemoveData(self, name):
        return self.fetch( self.data["remove"], name)
  def getAddDataTrustMe(self, name):
        return self.fetch( self.data["add"], name)[0]
  def getRemoveDataTrustme(self, name):
        return self.fetch( self.data["remove"], name)[0]

# This function will be called by the driver whenever a new shared data
# object is needed. It returns an object of the class type created
# above.
def  CreateSharedObject():
  return SharedInfo()
```

# Sample Behavior Script

The following script is supplied as a sample only and is not intended for deployment. Oracle Support does not provide support for this script.

```
#Title: Sample behavior script
Version = '1.0'
#IP Service Activator version: 7.0.0
#Date: 19-Mar-2008   IOS level: 12.1 - 12.2.8
#(c) T.Romain 2008
#This script is a sample generic behavior script
#
#begin preamble section
#N/A

######################################################################
#begin behavior section
#script_name = "do X"
#script_driver_type = "cisco"
#script_type = Device
#script_device_role = Access
#script_interface_role = Any
#script_apply_when = Before
#script_repeat = False
#script_apply_on_restart = True

#begin common section

#begin install section
class DoXer:
  def __init__( self ):
    self.reportingData={"id":_id, "failureCode":_failure_code}


  # The priority to execute this with: the lowest number will be
  # executed first.
  def priority( self ):
    return 2



  def Execute(self, parse):

    print "executing X script : priority "+ str( self.priority() )
# Retrieve data generated by pre-created data object
    interfaceData=_device.getThisCommitSharedData().getInterfaceAddData("XDATA")

    if( len(interfaceData) != 0 ):  # we have some interfaces
# Perform some function

      for iface in interfaceData.keys():
        print "Requested to put X on " + iface
    else:
      print "No interfaces to do X to"

# Finally insert an instance of the class into the shared area
_device.getLifetimeSharedData().insertBehavior( DoXer() )

#begin remove section
```

## Sample Data Script

The following script is supplied as a sample only and is not intended for deployment. Oracle Support does not provide support for this script.

```
#Title: Sample data script
Version = '1.0'
#IP Service Activator version: 7.0.0
#Date: 19-Apr-2008   IOS level: 12.1 - 12.2.8
#(c) T.Romain 2008
#This script is a sample generic data script
#

#begin preamble section
#N/A

#begin behavior section
#script_name = "iface wants X"
#script_driver_type = "cisco"
#script_type = Interface
#script_device_role = Access
#script_interface_role = Any
#script_apply_when = Before
#script_repeat = False
#script_apply_on_restart = True
#begin common section

data={
         "_reporting_id":_id,
         "_failure_code":_failure_code,



#begin install section
_device.getThisCommitSharedData().setInterfaceAddData(_
interface.getInterfaceName(),
         "XDATA", data)


#begin remove section
```

## Sample Controller Script

The following script is supplied as a sample only and is not intended for deployment. Oracle Support does not provide support for this script.

```
#Title: Sample controller script
Version = '1.0'
#IP Service Activator version: 7.0.0
#Date: 19-Apr-2008   IOS level: 12.1 - 12.2.8
#(c) T.Romain 2008
#This is a sample controller script
#

#begin preamble section
#N/A

######################################################################

#begin behavior section
```

```
#script_name = "controller"
#script_driver_type = "cisco"
#script_type = Device
#script_device_role = Access
#script_interface_role = Any
#script_apply_when = After
#script_repeat = True
#script_apply_on_restart = True

#begin common section
#begin install section
import string
import sre

tempbehaviors =_device.getThisCommitSharedData().fetchBehavior()
permbehaviors =_device.getLifetimeSharedData().fetchBehavior()
_device.openSession()
config= _device.deliverCommand("show running-config")
splitConfig=string.split(config,"\n")

_device.closeSession()


tmp=tempbehaviors.keys() + permbehaviors.keys()
tmp.sort()
lastI=-1
for i in  tmp :
  if( lastI!= i ):
    lastI=i
    if( permbehaviors.has_key(i) ):
      for j in permbehaviors[i]:  print j
        result=j.Execute( splitConfig )
        if(  result):
          print "FAILED!" #_
result.sendScriptObjectFailure(result[0],result[1],result[2])
    if( tempbehaviors.has_key(i) ):
      for j in tempbehaviors[i]:
        print j
        result=j.Execute( splitConfig )
        if(  result):
          print "FAILED"

#begin remove section
```