Cisco Application eXtension Platform Developer Guide

Revised: 7/7/08, OL-14813-01

The Cisco Application eXtension Platform Developer Guide provides information on how to create, package and install applications on the Cisco Application eXtension Platform (Cisco AXP).

Contents

This guide contains the following sections:

For more information on the following topics, see the *Cisco AXP User Guide*.

- Cisco AXP Hardware Requirements
- Cisco IOS Image Requirements
- Updating Cisco AXP Images
- Configuring the Cisco AXP Application Service Module
- Configuring the Application Environment

Cisco Application eXtension Platform Overview

The Cisco Integrated Services Router (Cisco ISR) is an integrated system within a single chassis. The Cisco ISR ties together and runs multiple value-added services such as voice, layer 2 switching, security, and application acceleration. In addition, integrated services can be hosted within the router's Cisco IOS software or the services can be decoupled and hosted on modular application service modules.

The Cisco ISR allows for blade hardware plug-in modules. These application service modules enhance the functionality, intelligence and flexibility of the router. The Cisco Application eXtension Platform (Cisco AXP) represents the next generation for this set of features.

Cisco AXP allows third parties such as system integrators, managed service providers, and large enterprise customers to extend the functionality of Cisco ISRs by providing their own value-added integrated services. On the service module, Cisco AXP hosts applications in a separate runtime environment with dedicated resources. In addition, Cisco AXP provides Application Programming Interfaces (APIs) that enable functions such as packet analysis, event notification, and network management to be utilized by hosted applications.

Cisco AXP has facilities and frameworks to host applications, and service APIs for integrating applications into the network. Cisco AXP provides the following features:

- Predictable and constant set of application resources.

  These resources (including CPU, memory, disk and network IO) are segmented, which ensures that the application and router features work independently, and without interference.

- Protection of the router and applications from rogue applications.

  In the event of an application crashing, other applications are not affected. The router will also continue running normally. This is achieved by having each installed application in its own virtual instance.

- Embedded Linux environment supporting the execution of applications written in the following programming languages: Java, C (native), Perl (interpreted), Python (interpreted), and Bash (interpreted).

  Native and interpreted applications written in other programming languages can be integrated by the third party application if the third party application uses additional support libraries and interpreters.

- Protection against running unauthorized software.

  Only our certified third parties can install software onto Cisco AXP.

- Robust debugging and troubleshooting facilities.

- Ability to modify the Cisco IOS configuration and obtain the status of Cisco IOS features via the provided APIs.

- Support of event notification.

  An application can receive the status of a Cisco ISR and take appropriate action.

- Integration of virtual devices.

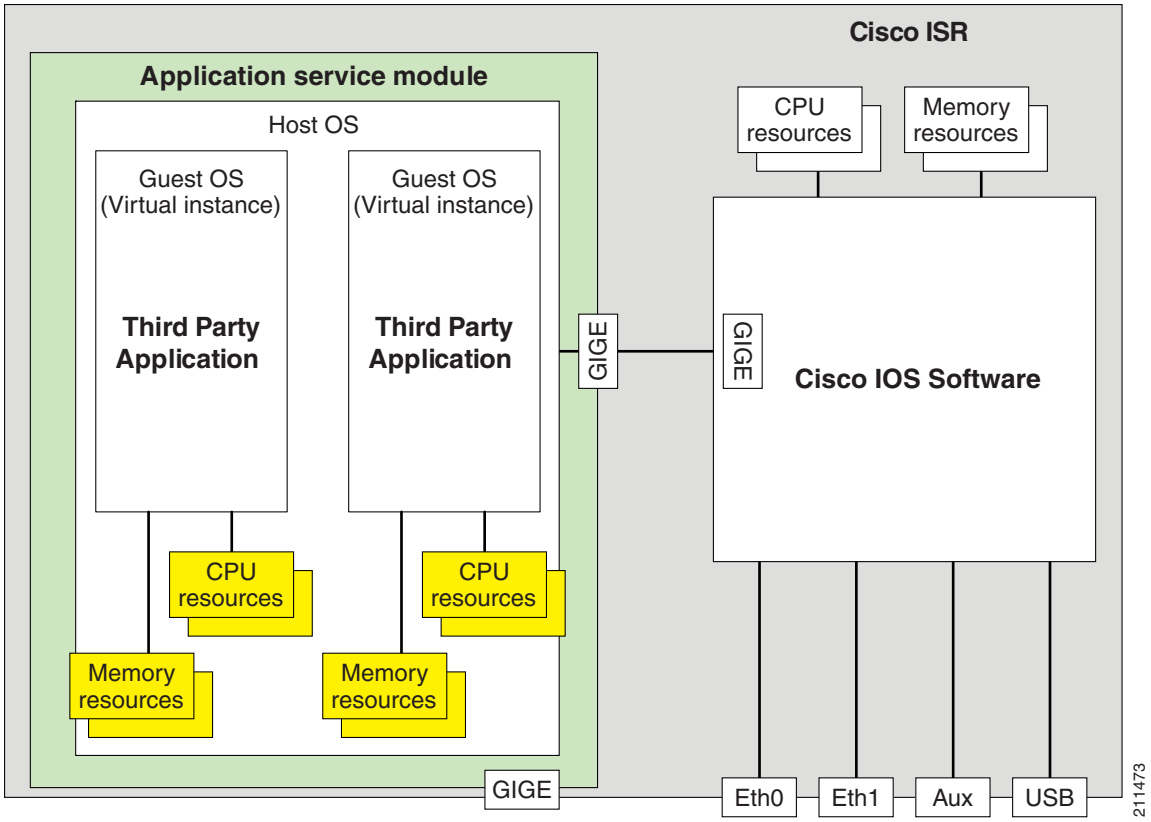
The Cisco IOS auxiliary serial port can be virtualized and appear in Cisco AXP OS as a local device. The application controls external peripherals attached to the router auxiliary serial port without special knowledge of the where the device is located.

- Firewall support.

  The Cisco AXP network interfaces are protected by a firewall for security. Ports can be opened using the "CLI Service API" section on page 18.

Figure 1 shows the relationship between applications in the Application Service Module and Cisco IOS within the Cisco Integrated Service Router (Cisco ISR).

Figure 1 Cisco ISR and Cisco AXP Service Module Interface



Cisco AXP Features

Cisco AXP provides a Linux-based hosting infrastructure, familiar to most application developers. Having a Linux base allows use of open source software for development. The hosting infrastructure installs and runs applications in separate security contexts based on Linux-VServer technology. These security contexts all use a common Linux kernel but have separate dedicated resources. Libraries can be independent in each security context. The network infrastructure is more secure by only allowing software that has been developed by partners approved by Cisco to be installed and run on the service module.

The key features of Cisco AXP are:

Note To return to the Table of Contents, click here.

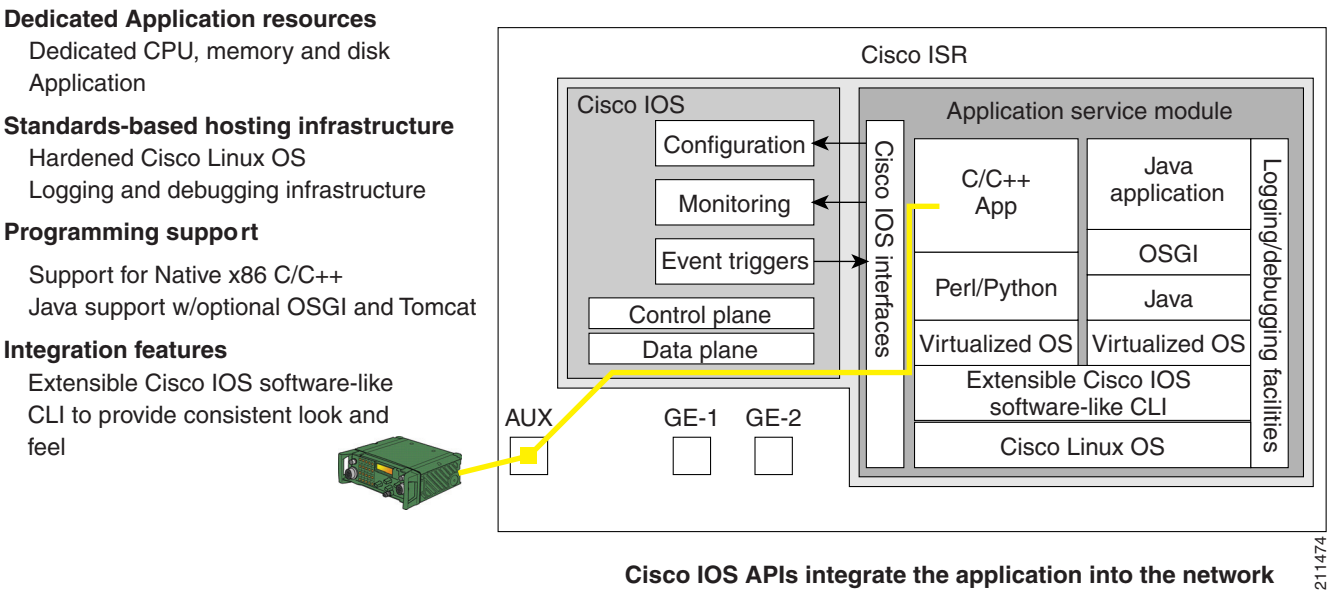
Hosting Infrastructure

Cisco AXP's Linux-based features allow applications to interact with Cisco IOS software. Program interfaces in Cisco AXP enable applications to modify Cisco IOS software configurations, receive notification of events from Cisco IOS software, and access peripherals attached to Cisco ISRs.

Applications can be tightly integrated with Cisco ISRs: this is a key advantage of using Cisco AXP for hosting, compared to using other server-based solutions. Cisco AXP hosts and executes an operating system in a virtualized environment.

The execution environment separates the application space from the router space, providing an extensible and flexible platform for hosting applications.

Figure 2 Cisco Application eXtension Platform on the Cisco ISR



Cisco AXP provides flexible management of Java applications through the OSGi framework.

The OSGi framework defines a standardized, component-oriented, computing environment for networked services, and enables the remote and secure life cycle management of Java applications. For more information on the OSGi specification, see the OSGi website at www.osgi.org.

Cisco AXP supports:

- Hosting of applications in native C/C++.
- Basic watchdog functionality to start, stop and monitor applications.
- Tools for compiling applications in an x86-based Linux environment.

Cisco AXP supports the following programming and scripting languages:

- Java
- C/C++
- Perl
- Python
- Bash

Virtualization

Cisco AXP uses virtualization at the operating system level to create an environment for multiple applications to run as stand-alone components.

Virtualization is achieved by partitioning resources to support concurrent execution of applications in multiple virtual machines/virtual instances.

Advantages of Virtualization in Cisco AXP

The advantages of virtualization in Cisco AXP are as follows:

Enhanced Security

Enhanced security due to process isolation.

Library Independence

Software that is ported to run on Cisco AXP may have dependency on Linux libraries that are not compatible with the library provided by Cisco AXP. To allow access to these Linux libraries, the application loads up its own guest operating system (guest OS) environment and therefore decouples the application from the Cisco AXP guest OS. Components of various Linux distributions can be installed on the same physical server, running in separate virtual instances. Each application has its own Linux operating system, utilities, and libraries. Each application has access to administrator privileges (root access) within its own virtual instance.

Resource isolation

Each application has its own set of CPU, memory, and disk resource limits. An application running in one virtual instance cannot use more resources than those allocated to that instance. This improves the stability of all the applications that are running in multiple instances. Virtual instances generally share disk space with the host OS, so disk space does not have to be preallocated.

Low Memory Overhead

A system using virtualization gains a higher performance with a low memory overhead.

Note Applications can only access the host OS by using Cisco AXP API calls.

Virtual Instances

In Cisco AXP, a virtual instance is a container within a virtual server. Process isolation gives security, and results from processes running in one virtual instance not being able to see processes running in another instance. This technology applies a change root barrier. If the application software is installed under `/home/app1` and then application processes start, the directory /`home/app1` becomes the root directory for the application processes. The processes can only access directories below this root directory and do not have permission to escape from `/home/app1`.

User level processes are virtualized: one Linux kernel is used throughout the system, which uses a low amount of memory and has a low scheduling overhead with high speed IO access. The system provides near native OS performance.

Application software (which can contain multiple processes and functionalities) is packaged into the Super Lightweight Installer Mechanism (SLIM) format by using the Cisco AXP packaging tool. The package is installed into the virtual instance's root directory: the package runs within one virtual instance.

An application running inside a virtual instance is provided with its own operating system environment called the guest OS. A host OS manages all the guest operating systems. An application can run as root in a guest OS.

Files delivered by a third party cannot be shared between virtual instances. Cisco AXP does not support applications in different virtual instances having read or write access to a file in a shared area.

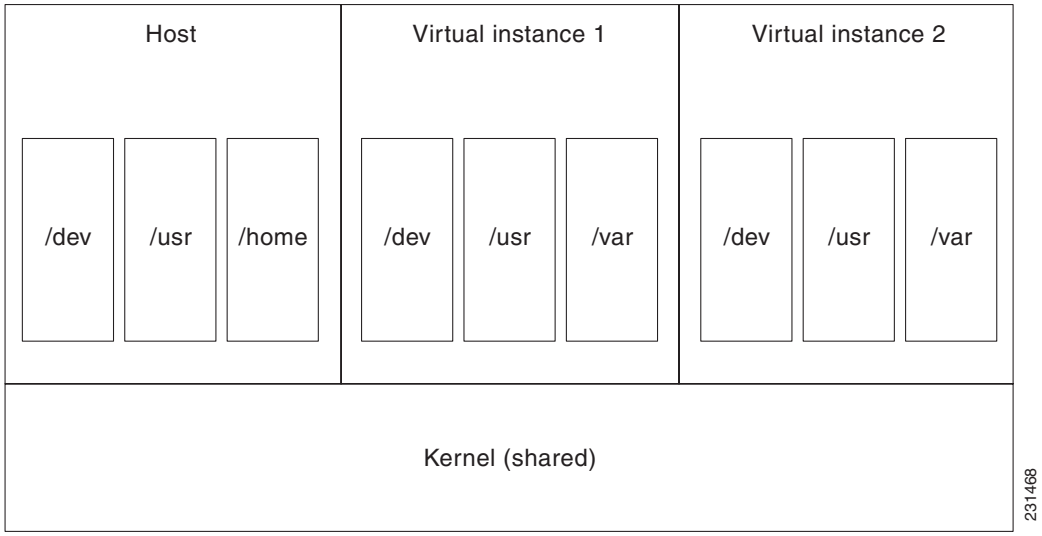
To further improve security, the shell environment of the host OS is not available by default. Only the CLI within the guest OS is provided to manage the operating environment. Shell access to the guest OS is controllable by the application developer.

Devices are not virtualized and are shared across all virtual instances. Therefore, IO operations do not incur additional overhead.

Network interfaces cannot be virtualized, due to limitations in the Linux-Vserver. However, a "virtualized loopback interface" mechanism exists, as explained in Loopback Interfaces, page 7.

Interface hiding occurs in a virtual instance. Network routing tables are not virtualized; however, multiple routing tables can be configured in Cisco AXP.

Figure 3 Cisco AXP Virtualization



Virtualization is achieved by kernel-level isolation, allowing multiple virtual instances to run simultaneously. Virtual instances are isolated by the shared kernel, allowing each instance to utilize common system resources.

Note Cisco AXP's virtual instance model does not allow third parties to independently install kernel modules or device drivers.

Loopback Interfaces

Due to limitations in Linux-Vserver, network interfaces cannot be virtualized.

An application can use a loopback to communicate internally without worrying that an interface is being used by other virtual instances or the host.

Cisco AXP provides a way to virtualize loopback interfaces for each instance.

For example, a virtual interface of the loopback (lo:2, lo:3) and its corresponding address (127.0.0.2, 127.0.0.3) is provided to a virtual instance.

The default loopback (127.0.0.1 and lo) is reserved for the Cisco AXP host loopback interface, and is also accessible in each virtual instance.

The /etc/host file of each virtual instance is populated with the virtualized loopback interface and the Cisco AXP host loopback interface. For example, if an application is installed to Cisco AXP, its /etc/host file should look like this:

```
127.0.0.x         localhost.localdomain localhost
127.0.0.1         apprehost
```

If you issue the command **netstat** in the virtual instance, this shows that "lo" is associated with "127.0.0.1", and "lo:x" is associated with "127.0.0.x".

x is a number ranging in value from 2 upwards, depending on the number of virtual instances.

This number is dynamically allocated, so applications must not assume that x stays the same.

For example, when a new application is installed, the new application can be assigned x while the original application can be assigned x+1. Applications should instead rely on “localhost” being their own virtualized loopback interface.

127.0.0.1 is assigned to “apprehost” and 127.0.0.x is dynamically assigned to “localhost”.

You should be aware of this when dealing with a loopback interface in your application.

Note The following is a potential security issue: The virtualized loopback interface for each virtual instance can be accessed by other virtual instances.

Dedicated Application Resources

Cisco AXP provides a predictable and constant set of resources such as CPU, memory, and disk space. These resources are segmented to allow an application to run on the Cisco AXP service module without affecting the performance of other router features.

You can specify CPU, Memory, and Disk space limit (specified in MB), using the CLI with administrator privileges.

Management of CPU and Memory by Cisco AXP is summarized in the following sections:

- CPU Resources, page 8
- Memory Resources, page 8

Note To return to the Table of Contents, click here.

CPU Resources

A service module has available CPU resources that are specified as a CPU index. The CPU index of a service module is a value that is relative to a base value of 10000 assigned to the following configuration: 1.0 GHz Celeron M CPU on the application runtime engine of an NME_APPRE_302-K9 service module.

For example, the CPU index for the service module AIM_APPRE 102 (blade) is 3000.

When your application package is installed, it will use approximately the same amount of CPU resources on either an NME or AIM service module. The CPU Index of each service module is shown in Table 1.

Table 1 Service Module CPU Index

Service Module	CPU Index	CPU Resources Required for Host OS	CPU Resources Remaining
AIM APPRE 102	3000	300	2700
NME APPRE 302	10000	300	9700
NME APPRE 522	14000	300	13700

Memory Resources

Cisco AXP does not use disk swapping for virtual memory. Therefore, the amount of memory available to an application is limited by the physical memory of the system.

In the context of virtual instances, the memory limit specifies the maximum memory available for each virtual instance. When memory overcommit is enabled, if the total memory usage within the virtual instance exceeds the memory limit (specified in MB), then processes within the virtual instance are killed.

Application Packaging

Application Types

Cisco AXP supports a range of applications and programming languages, including applications written in Java, C, and Bash scripts. Virtual environments in Cisco AXP have an embedded Cisco Linux distribution with add-on libraries and packages.

Packaging

In order to install an application into a virtual instance, you must package your application.

The packaging tool allows a third party to package, bundle and sign their application package. The packaging tool requires that Cisco assigns the third party a developer's certificate enabling the third party to develop applications. The third party owns a private key that is used to sign the package.

The system and application packages include the following:

- **Core System Package**

  The core system package is preinstalled on all Cisco AXP service modules, and is necessary to support applications and virtualization.

  The Cisco AXP operating system includes virtualization support, kernel and system utilities such as Linux commands, Router Blade Communication Protocol (RBCP), and a CLI server.

  The host Operating System (host OS) consists of a kernel of the core system package that is shared by all virtual instances running on the service module.

  The guest Operating System (guest OS), also known as a hosted operating system, is an embedded Linux OS that shares the kernel of the host OS.

  The guest OS includes virtual environment setup and management utilities, and software required to support installation of the application. The guest OS shares the Linux kernel of the core system package in the host OS.

  For a list of commands available in a guest OS, see the "Appendix B: Commands in Guest OS" section on page 124.

  Packages can be divided into the following two types:

- **Infrastructure Add-on Packages**

  Infrastructure add-on packages extend the capabilities of the guest OS (see Core System Package above). The packages may contain both components developed by us and open source software components. Infrastructure add-on packages are created and signed by us: they cannot me modified by a developer working for a third party. Infrastructure add-on packages can be selectively included with your application software during packaging.

  Cisco AXP supports a broad range of application hosting environments. Packages are separated based on their intended application. For example, a package intended for OSGi development needs the OSGi infrastructure add-on package, but it does not need to include the Tomcat package.

  Files that are included in infrastructure add-on packages are shared on a read-only basis between multiple virtual instances.

  Examples of infrastructure add-on packages:

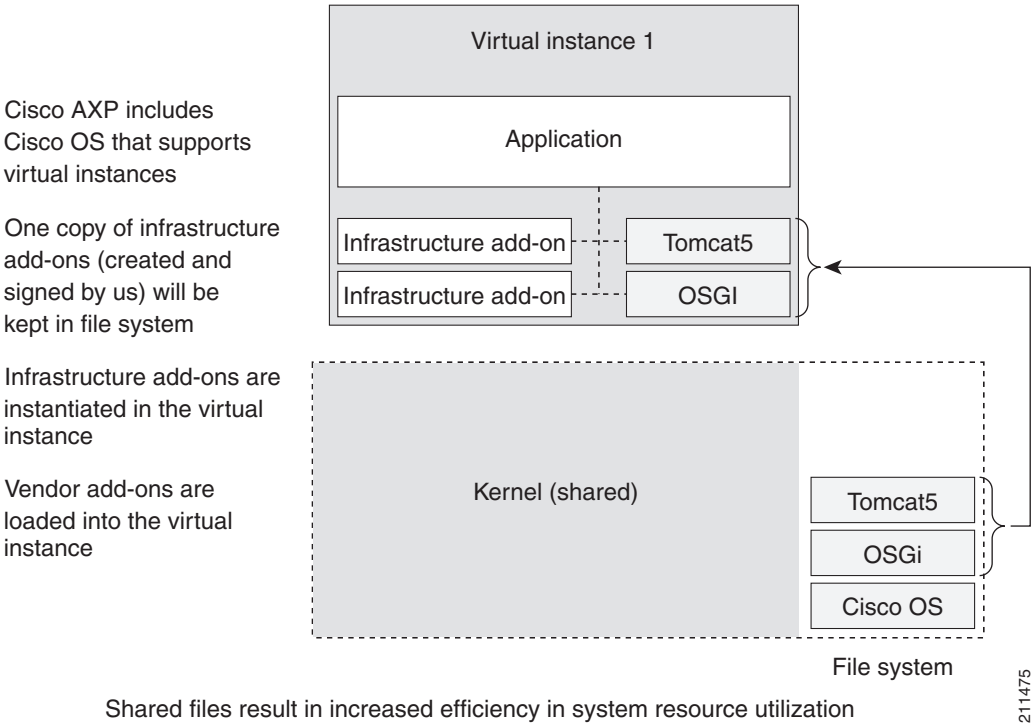
- Cisco IOS Service API for accessing Cisco IOS.
- OSGi environment and supporting libraries.
- Perl Scripting Language for supporting Perl interpreter.
- Debug package.

- **Hosted Application Packages**

  Hosted application packages are created and packaged by third party developers. These packages contain binaries, supporting libraries, and any data files required to run the third party's software applications. Each application package is signed using a developer's private key and verified using the respective developer's certificate.

  The binaries in these packages can only be hosted in virtual instances of the Cisco AXP system.

Figure 4 Cisco AXP Add-on Packages



Shared files result in increased efficiency in system resource utilization

Cisco AXP ships preloaded with our operating system that supports multiple virtual instances. One copy of an infrastructure add-on package is stored in a file system created and signed by us. An example of an infrastructure add-on package is Apache Tomcat.

Copies of an infrastructure add-on package are instantiated in virtual instances during postinstallation. An infrastructure add-on package exists in the file system, but is shared across virtual instances. This compares with a third party add-on package, which is a package containing the third party application inside one virtual instance. There is only one virtual instance for each third party application. A bundling tool is also provided in the SDK to compile multiple packages into a single bundle.

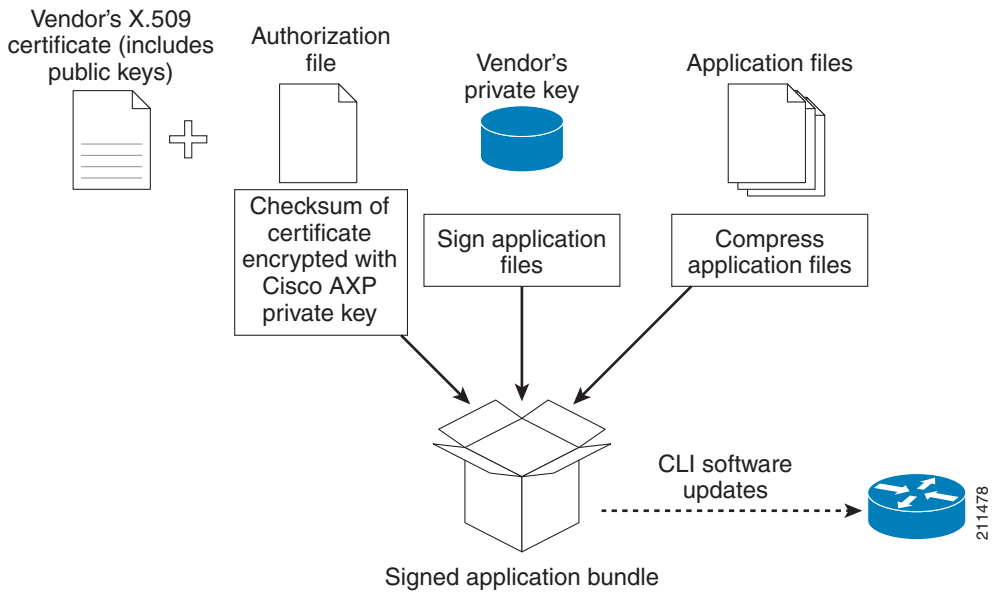
Cisco AXP Certificates

Cisco AXP uses cryptographic signatures to verify packages that are signed by us to prevent unauthorized software to be loaded into the operating system.

This functionality is extended to applications developed and packaged by third parties. Third parties will not have access to our private keys for package signing and must manage their own public or private key pairs.

Figure 5 Cisco AXP Application Packaging

Application packaging



We manage all permissions for installing software on Cisco AXP. Permission to install application third party software is granted by providing the application third party with a checksum of the third party's X.509 certificate. See Figure 6 to see the flow of events in obtaining permissions.

Figure 6 Cisco AXP Certificates

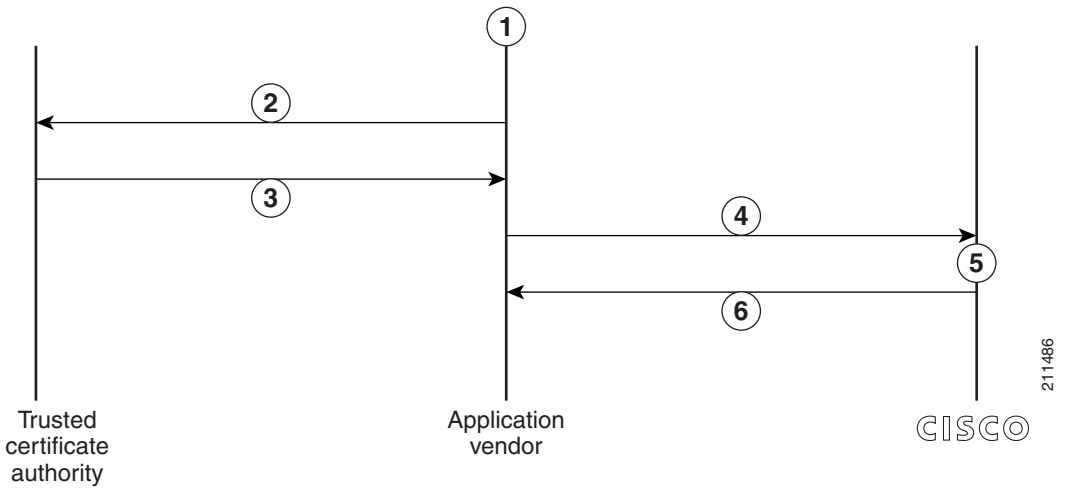


Table 2 Steps in the Development Authorization Process

Step	Description
1	Third party generates a certificate request (private/public keys).
2	Third party requests a signed certificate from the certificate authority.
3	Certificate authority responds to the third party request, by providing a signed X.509 certificate[1].
4	Third party requests software development authorization from Cisco (includes signed certificate from certificate authority).
5	Cisco authorizes the signed certificate.
6	Cisco responds with a software development authorization certificate[2].

1. The third party is responsible for providing the signed X.509 certificate to Cisco. (Step 3)
2. The software development authorization certificate is a file that contains an encrypted checksum authorizing the third party to install software in Cisco AXP. The authorization certificate only applies to an X.509 certificate signed by the certificate authority (Step 3 in Table 2.).

Notes on Steps in the Authorization Process

During application packaging, the third party includes the following items in the package: Cisco authorization certificate, third party certificate, third party private key, and third party application.

The checksum of the authorization certificate is checked before the authorization certificate's public key is used to verify the application. The certificate can be used to verify that the application files are installed with the application package. The authorization certificate is held in the application's manifest file and stored outside the virtual environment of the host OS.

If the target hardware contains a device to securely store cryptographic keys or certificates, then Cisco AXP does not use the third party's key to verify software integrity. The installed software is re-signed using a private key tied to the hardware and is verified using the corresponding public key.

The Cisco AXP software development kit (SDK) provides third parties with the tools for packaging and signing their application files, and for attaching their certificate to the application bundle.

Running Applications on the Guest Operating System

An embedded Linux guest OS is automatically included in the application software dependency list. The guest OS contains a set of basic Linux binaries and libraries. The libraries are listed in "Appendix C: Libraries in Guest OS" on page 125. The guest OS shares its kernel with the host OS and is based on the Linux Standard Base specification. After being installed on the service module, every application instance has access to its own guest OS and may add or modify its components.

In the following sections, starting an application is explained in "Application Startup" on page 13, and monitoring the status of an application is described in "Application Status Monitoring" on page 16.

The Guest OS is further explained in the following sections:

For information on libraries and commands available in the guest OS, see the "Appendix C: Libraries in Guest OS" section on page 125 and "Appendix B: Commands in Guest OS" on page 124.

Note To return to the Table of Contents, click here.

Application Startup

Application startup is either an automatic or manual process:

Automatic Startup

For automatic startup, perform the following steps.

Step 1 Create a startup script in directory /etc/rc.d/init.d

Step 2 In directory /etc/rc.d/rc3.d, reference the startup script using a startup softlink.

```
S<priority><app-name>
```

,where priority is 0–99, and app-name is the name of your startup script in directory /etc/rc.d/init.d

Step 3 In directory /etc/rc.d/rc6.d, reference the startup script using a kill softlink:

```
K<priority><app-name>
```

,where priority is 0–99, and app-name is the name of your startup script in directory /etc/rc.d/init.d

The application starts.

Manual Startup

Note For manual startup, first obtain access to the guest OS shell.
To be able to use **linux shell**, first perform the steps in Console Access using a Debug Package as a Dependency, page 14.
To be able to use **connect console**, first perform the steps in Console Access using a Postinstall Script, page 14.

For manual startup, perform the following steps.

Step 1 **app-service <name>**

<name> is the application's name.

Step 2 Depending on the displayed command(s), enter one of the following commands:

- **linux shell**
- **connect console**

You will be placed in the guest OS shell from where you can start your application.

Obtaining Console Access for the Application

To obtain console access for your application choose one of the following procedures:

- Console Access using a Debug Package as a Dependency, page 14
- Console Access using a Postinstall Script, page 14

Console Access using a Debug Package as a Dependency

To obtain console access using a debug package as a dependency, perform the following steps.

Step 1 Check that the application development package axp-app-dev.<platform>.<version>.pkg is available. When you are packaging your application, specify the application development package as a dependency.

Step 2 Install your application into Cisco AXP.

Your application has console access to the guest OS shell via the command **linux shell**.

Console Access using a Postinstall Script

To obtain console access you create two scripts before installing your application. The first script is a login script, linked to /bin/console using a second script: the postinstall script.

To obtain console access using a postinstall script, perform the following steps.

Step 1 Create a login script. For example, create a file /source/bin/login.sh , where /source is the build directory.

login script: Example

```
#! /bin/sh
/bin/bash --login
```

Step 2 Create a postinstall script. For example, create script /source/bin/postinstall.sh , where /source is the build directory.

postinstall script: Example

```
#! /bin/sh
ln -s /bin/login.sh /bin/console
```

When you are packaging your application, see the postinstall script. Change the permissions of the postinstall script for non-owner users to read-execute:

```
chmod 755 /opt/tcptrace/postscripts/postinstall.sh
```

By using the postinstall script, you allow your application to have console access to the guest OS shell via the command **connect console**.

Application Status

An application must invoke the notification command to signal its status. See the "Notification" section on page 15. For example, this is necessary in order for configuration CLI commands to detect when the application is available.

An application that sends a status code indicating that it is up and running, expects configuration commands to arrive at any time. The possible status codes are shown in Table 3.

Table 3 Status Codes

Status Code	Description
INITIALIZING	Application is starting up.
ALIVE	Application is running.
DOWN	Application is down.

Notification

Notification command location: /bin/app_status_notifier.

Application name and status code are passed as parameters or arguments to the notification command.

Notification: Example:

```
#/bin/app_status_notifier myapp ALIVE
```

Viewing an Application's Health

To view the health of an application perform the following steps.

Step 1 **app-service** <name>

<name> is the application name.

Step 2 **show state**

For further information, see "Viewing the Application State" in the *Cisco AXP User Guide*.

Application Status Monitoring

An application must provide one or more watchdog scripts or executable files bundled in their package to use the Cisco AXP application monitoring feature.

Two methods of monitoring the application are described in the following sections.

- Watchdog Scripts, page 16
- Status Monitor, page 17

Watchdog Scripts

If the application state is "online" and application health is not "DOWN", the application is monitored by the application status monitor.

The application must provide one or more *watchdog scripts* (also known as health check scripts or health check executables). Shell scripts and C executables are supported. The number of scripts is application dependent.

Bundle the watchdog scripts in the application package.

An application has its own way of determining if another application is running. For example, an application may check the Process Identifier (PID), or check the response to a "ping".

Watchdog scripts must be placed in the predefined directory /opt/app_status_monitor/watchdogs/

The watchdog scripts directory is in the application's vserver directory, and needs to have executable permissions. All watchdog scripts inside that directory will be invoked, and if you want to ensure the correct ordering for script invocation, you can follow the following template for naming a watchdog script:

```
W**<name>.sh
```

, where ** is a two digit number. For example, one watchdog script is called W01myapp.sh.

The application status monitor has a heartbeat of 5 seconds, where 5 seconds = 1 interval. Five seconds is the minimum interval that the application status monitor requires for monitoring.

When any one watchdog script returns a non-zero status code, information for this failed watchdog will be logged, including: watchdog name, return status, and time of failure. A recovery counter counts how many times that failure occurs, and works as a delay in taking any action. A recovery counter of 3 means that the application monitor has run the monitoring for 3 iterations and is either getting non zero return status, or the watchdog has been running for over 3 monitor intervals and is not returning. The configurable recovery threshold determines the number to which the recovery counter increments to before taking the next action. When the recovery threshold is reached the virtual instance is restarted. After the virtual instance is restarted, application status monitoring continues, and the monitoring steps above continue. There is no limit on how many times the virtual instance restarts.

To configure the monitor interval, see the "Status Monitor" section on page 17. Also see the *Cisco AXP User Guide*. Each script or executable needs to return a status code.

- status code = zero (0): application is healthy and alive.

- status code = non-zero value: application is not functional. For example, the application has crashed.

Watchdog Script: Example:

```
#!/bin/bash
APP=test.sh
APPNAME_NO_EXT=test
PID_FILE=/var/run/${APPNAME_NO_EXT}.pid
if [ ! -e $PID_FILE ]; then
exit 1;
fi
PID_FROM_FILE=`cat ${PID_FILE}`
for x in `ps -ef|grep $APP |awk '{print $2}'`
do
if [ $x == "${PID_FROM_FILE}" ]; then
exit 0
else
exit 1
fi
done
```

In this example watchdog script, the startup script must have populated the /var/run/<app>.pid file with the pid of the currently running application. The startup script should also account for various scenarios such as:

- allowing only one instance of the application to run
- removing the /var/run/<app name>.pid file on shutdown

When any watchdog script returns a nonzero status code, information is logged such as name of watchdog, return status, and the time of failure. A recovery counter counts how many times failures happen. A recovery counter value of 3 indicates one of the following two conditions:

- Application status monitor ran three times and received non-zero status code
- Watchdog ran over 3 monitor intervals and did not return a non-zero status code

There is a configurable recovery threshold that decides the value that the recovery counter reaches before taking the next action. When the recovery threshold is reached, the virtual instance is restarted.

After the virtual instance restarts, the application status monitor continues to run, and the above two conditions are tested again. There is no limit on how many times the virtual instance restarts.

Status Monitor

The following example commands include **show status-monitor**, and commands to configure the monitor interval and recovery threshold. For more information, see The Status Monitor in the *Cisco AXP User Guide*.

- Show status monitor

```
blade# app-service myapp
(myapp)# show status-monitor
Application:                    myapp1
Monitor status:                 PASSED
Monitor in progress:            No
Last executed watchdog:         W01myapp1_test.sh
Last executed date:             Tue Jul 10 10:22:06 PDT 2007
Last failed watchdog:           W01myapp1_test.sh
Last failed return code:        4
Last failed date:               Mon Jul  9 12:34:18 PDT 2007
Last restarted date:            Mon Jul  9 12:33:32 PDT 2007
Recovery threshold:             6
Monitor interval:               2
```

- Change the monitor interval and recovery threshold

```
blade# config terminal
(config)# app-service myapp
(config-myapp)# status-monitor monitor_interval 24 recovery_threshold 10
(config-myapp)# end
```

Configuration File

In addition, you can provide a default configuration by using a configuration (config) file that is packaged together with your application.

The config file name and path:

```
/opt/app_status_monitor/config/config
```

Each record in the config file is in the following format:

```
[monitor_interval | recovery_threshold] [1-99]
```

The default value monitor_interval is 12. The application is monitored every 12 heartbeats. If the monitor interval is 12, then the monitor is active on each virtual instance every minute.

The default value for recovery_threshold is 5.

Note Both monitor_interval and recovery_threshold must be present in the config file.

config file: Example

```
monitor_interval 50
recovery_threshold 10
```

Configuration values, (such as the values of 24 and 10 set in the Change the monitor interval and recovery threshold example above), take precedence over the config file values. The config file only serves as a source of default values when the application is installed.

CLI Service API

The CLI service API allows your application to interact with the underlying CLI server on the host.

For example, Java application code can issue EXEC mode commands or configuration mode CLI commands.

Care must be taken to issue commands suitable for a particular mode. For example, issuing **show run** in EXEC mode, otherwise an error occurs. The main supported programming/scripting languages are C, C++, Java, Python, and Perl. The CLI service APIs for these languages are shown in the following sections.

CLI commands can be submitted in batches by the application within a single instance. This is useful in automating repetitive tasks such as reconfiguration after an image refresh.

The Cisco AXP SDK contains the libraries and modules required for writing CLI service EXEC mode or configuration mode commands. The languages supported by the SDK and the associated libraries/modules are listed in Table 4.

Table 4 Files for CLI Service API

Language	Library/Modules/Header File
C/C++	/include/appreapi.h /lib/libappreapi.so
Java	/jar/appreapi.jar
Python	/python2.3/AppreAPI.py
Perl	/perl/AppreAPI.pm

CLI Service APIs

Java API

```
public int exec (AppreMessage msg);
public int config (AppreMessage msg);
```

The Java bean object AppreMessage contains both the request and response string.

After the code has run, the returned status code indicates completion/failure. For status code values, see Table 5.

```
public class AppreMessage{
    private String _response;
    private String _request;

    public void AppreMessage(){
        this._response = "";
        this._request = "";
    }
    public String getRequest(){
        return this._request;
    }
    public void setRequest(String request){
        this._request = request;
    }
    public String getResponse(){
        return this._response;
    }
    public void setResponse(String response){
        this._response = response;
    }
}
```

C/C++ API

Note The calling program is responsible for freeing allocated memory. If the calling program sets the char* to NULL, the system allocates the required memory, but the calling program must still free the memory.

```
    int appreapi_exec(AppreMessage_t* msg);
    int appreapi_conf(AppreMessage_t* msg);

    free (msg.response);
```

AppreMessage_t is a struct containing the request and response string.

After the code has run, the returned status code indicates completion/failure. For status code values, see Table 5.

```
typedef struct AppreMessage_t{
    char* request;
    char* response;
    int size;
} AppreMessage_t;
```

Perl API

```
sub exec (request)
sub config (request)
```

The request string is input and the status code value is output. For status code values, see Table 5.

Python API

```
sub exec (request)
sub config (request)
```

The request string is input and the status code value is output. For status code values, see Table 5.

Bash API

```
appreapi [--mode name] [[--file filename] | [cmdlist]]
```

The request string is input and the status code value is output. For status code values, see Table 5.

Table 5 CLI Service Return Status Codes

Status Code (C/C++)	Status Code (Java)	Value	Description
APPRE_OK	AppreAPI.OK	0	Request successfully processed.
APPRE_FAIL	AppreAPI.FAIL	1	Request aborted because of an error. If a response string is returned (Java/C), it contains a description of the error.
APPRE_FILE	AppreAPI.FILE	2	Request processed. If a response string is returned (Java/C), it contains the filename and path, and is written to a file.

CLI Service API Examples

In the following examples, commands are issued by passing a string to a function/method such as `request` (for C) or `setRequest` (for Java). More than one command may be passed at one time, by separating each command with a comma. For example, to use setRequest( ) to call two or more system commands:

```
msg.setRequest(cmd)
```

where `cmd` is a string containing a sequence of commands separated by commas.

For example,

```
String cmd = "show run,show trace”
```

The following sections show code fragments that implement AppreMessage in the main programming/scripting languages of C/C++, Java, Perl, and Python.

- C/C++: Example, page 21
- Java: Example, page 21
- Perl: Example, page 22
- Python: Example, page 22

C/C++: Example

```
#include "appreapi.h"

int main(int argc, char** argv){
    char buf[] = "show run";
    AppreMessage_t msg;
    msg.size = 0;
    msg.response = NULL;
    msg.request = buf;
    int status = APPRE_FAIL;
    status = appreapi_conf(&msg);
    status = appreapi_exec(&msg);
    return status;
}
```

Java: Example

```
import com.cisco.aesop.apphosting.appreapi.*;
```

```
public static void main (String[] args){
    int status = 0;
    CommonServiceImpl apiCall = new CommonServiceImpl();
    AppreMessage msg = new AppreMessage();

    msg.setRequest("show run");

    status = apiCall.exec(msg);
    status = apiCall.config(msg);
}
```

In the above example the Java class CommonServiceImpl implements the API access service.

The class has two methods for exec and config mode:

```
public int exec (AppreMessage msg)

public int config (AppreMessage msg)
```

Perl: Example

```
use AppreAPI;

$api = new AppreAPI::AppreAPI();
$req = "show run";
($val, $res) = $api->exec($req);
($val, $res) = $api->config($req);
```

Python: Example

```
from AppreAPI import *

api=AppreAPI()
status, response = api.exec(request)
status, response = api.config(request)
```

Logging Support

Cisco AXP provides logging support for applications that include a syslog or log4j logging facility, and file rotation of logging files. Different logging levels can be set. If an error occurs with a level below the logging level, the error is not reported in a log file. For more information on logging errors, see the "Logging" section on page 87.

SNMP

Use SNMP in your application by using net-snmp code. There are two locations for net-snmp code:

- C library in the guest OS: /lib/libsnmplib.so
- Header files in the Cisco AXP SDK within directory ~/include/net-snmp. The net-snmp library uses a socket for internet access.

The net-snmp API is described at http://net-snmp.sourceforge.net/docs/man/.

Net-snmp Tutorial and Example Links

A net-snmp tutorial and sample application can be found at http://net-snmp.sourceforge.net/.

For example code that uses net-snmp API calls, see "Net-snmp: Example" section on page 108.

Infrastructure Add-on Packages

For detailed descriptions of the value-added service APIs or add-on packages that are available, see the List of Infrastructure Add-on Packages below.

Package Name

Package names have the format: <package>.<platform>.<version>.pkg

where, for example, package axp-tomcat5.nme.1.0.1.pkg has package = tomcat, platform = nme, and version = 1.0.1

The version number in this example, is the same as the version number in the main Cisco AXP package (axp-k9.<platform>.<version>.pkg).

Note that payload files for each infrastructure add-on package are available for download, with suffix ".prt1".

Downloading the Infrastructure Add-on Package

To use an infrastructure add-on package, download the package from www.cisco.com and bundle it with your application, before installing your application on the service module.

List of Infrastructure Add-on Packages

The following infrastructure add-on packages are provided in the Cisco AXP.

Note To return to the Table of Contents, click here.

OSGi

Package name: axp-prosyst-mbs6.<platform>.<version>.pkg

Package name: Example

axp-prosyst-mbs6.nme.1.0.1.pkg

The OSGi framework specification defines how bundles of Java applications or components can be remotely installed, started, stopped, updated and uninstalled without impacting the operation of the device.

The OSGi Infrastructure add-on package bundling allows applications or components to be remotely started, stopped, updated, or uninstalled without having to restart a computer operating system.

ProSyst is a commercial OSGi software provider and the ProSyst package provides an embedded system with extensions and optimization in an OSGi™ framework.

The package axp-prosyst-mbs6.<platform>.<version>.pkg is the ProSyst Embedded Server mBS6.0 Framework package.

If the package axp-prosyst-mbs6.<platform>.<version>.pkg does not meet the requirements of your system, purchase a OSGi™ bundle from ProSyst and package it with your application.

The supplied infrastructure add-on package axp-prosyst-mbs6.<platform>.<version>.pkg does not have an embedded licence key. Before using the package, obtain a license from ProSyst. Put the license key file (domain.crp) in your project directory /opt.

The startup script for axp-prosyst-mbs6.<platform>.<version>.pkg firstly copies the license key file before starting the mBeddedServer.

Manage the OSGi framework using the Cisco AXP CLI command: **connect osgi.** See the "Configuring the Application Service Environment" section of the *Cisco AXP User Guide*.

This command cross connects to the text console of the Prosyst OSGi framework. From the text console you can manage the OSGi framework using Prosyst text commands (CLI).

Tomcat

Package name: axp-tomcat5.<platform>.<version>.pkg

Package name: Example

axp-tomcat5.nme.1.0.1.pkg

Use the tomcat package to embed the Tomcat 5.5 servlet container in a virtual instance.

Application SSH

Package name: axp-ssh-4.6p1-k9.<platform>.<version>.pkg

Package name: Example

axp-ssh-4.6p1-k9.nme.1.0.1.pkg

The application SSH package is used in production for SSH Tunneling, see "SSH Tunneling" section on page 102.

Application Development Package

Package name: axp-app-dev.<platform>.<version>.pkg ; for example, axp-app-dev.nme.1.0.1.pkg

Access the Linux shell to issue CLI commands, see "Obtaining Console Access for the Application" section on page 14.

Add either package axp-ssh-4.6p1-k9.<platform>.<version>.pkg or axp-app-dev.<platform>.<version>.pkg as a dependency when you package your application.

The application development package provides an SSH server that runs in a virtual instance.

SSH Tunneling, rsync utility, and GDB debug are described in Debugging Tools, page 102.

CLI Plug-in Distribution Service

Package name: axp-cli-plugin.<platform>.<version>.pkg ; for example, axp-cli-plugin.nme.1.0.1.pkg

The Cisco AXP CLI Plug-in distribution service is an infrastructure add-on package that provides the necessary libraries that allows you to use your CLI within the Cisco AXP system. The service distributes CLI plug-ins from the Cisco AXP CLI server to the application.

There are three main components you need to define: CLI plug-in definitions, CLI plug-in distribution listener and CLI plug-in actions. See the "CLI Plug-in Applications" section on page 56.

Cisco IOS Service API

Package Name: axp-iosapi.<platform><version>.pkg ; for example, axp-iosapi.nme.1.0.1.pkg

The Cisco IOS Service API allows you to write applications that access router information and change system configurations using commands equivalent to Cisco IOS commands in config and EXEC modes.

The Cisco AXP SDK contains the libraries and modules you require for writing applications that use Cisco IOS configuration mode commands. The languages supported by the SDK and the associated libraries/modules are listed in Table 6.

Table 6 Files for Cisco IOS Service API

Language	Library/Modules/Header File
C/C++	iosapi.h libiosapi.so
Java	iosapi.jar
Python	iosapiPython.so IosapiFactory.py Iosapi.py
Perl	iosapi.pm
Bash	iosapi

Each of the APIs listed in the following section, includes a timeout API, which you can use to prevent the calling program from hanging (getting stuck) while waiting for a return message. The length of time of the timeout is passed as an argument to the timeout API. The calling program can have this timeout period set up for a particular execution of the batch. The timeout applies to all subsequent Cisco IOS software commands that follow the call to the timeout API, during one session.

Cisco IOS APIs

APIs for accessing Cisco IOS commands are listed below.

C/C++ API

```
typedef struct IosServiceAPI_t {
    apiMethod    exec;
    apiMethod    config;
} IosServiceAPI_t;

extern int getIosApi(const char *serviceName, IosServiceAPI_t* iosapi);

void set_timeout (int sec)
```

Java API

```
public int exec(IosapiMessage msg);

public int Config(IosapiMessage msg);

public void set_timeout (int sec);
```

Python API

```
class Iosapi
def Config(self, request)

def Exec(self, request)

def Set_timeout(self, sec)
```

Perl API

```
package Iosapi::Iosapi;
sub exec(self, request);

sub config(self, request);

sub set_timeout (timeout)
```

Bash API

```
appreapi [--mode name] [[--file filename] | [cmdlist]] [--timeout sec]
```

Cisco IOS API Response and Status

In the case of C/C++/Java, the API returns a response string, which is stored in the message struct.

In the case of Perl and Python, the API returns a response string and a status code (multiple return values).

The possible return status codes are explained in the table below.

Table 7 Cisco IOS Return Status Codes

Status Code (C/C++)	Status Code (Java)	Value	Description
IOSAPI_OK	IosServiceAPI.OK	0	Request processed and successfully executed.
IOSAPI_FAIL	IosServiceAPI.FAIL	1	Request aborted because of an error. If a response string is returned (Java/C), it contains the error message.
IOSAPI_FILE	IosServiceAPI.FILE	2	Request processed successfully. If a response string is returned (Java/C), it contains the file name and path, and is written to a file.

Cisco IOS API Examples

In the following Cisco IOS API examples, commands are issued by passing a string to a function/method such as `request` (for C) or `setRequest` (for Java). More than one command may be passed at one time, by separating each command with a semi-colon. For example, to use `setRequest` to call two or more system commands:

```
msg.setRequest(cmd)
```

where `cmd` is a string containing a sequence of commands separated by semi-colons.

For example,

```
String cmd = "int fa0/0; ip address 172.31.100.195; no shutdown"
```

C/C++: IosapiMessage_t

For C/C++, use the structure type variable IosapiMessage_t.

```
typedef struct IosapiMessage_t {
    char   *request;
    char   *response;
    int     size;
} IosapiMessage_t;
```

`response`: set to zero, for the API to allocate memory for the response, or set to non-zero to allocate memory for the response yourself.

`size`: set to zero: if the API allocates memory, or set to non-zero: if caller allocates memory.

In both cases you need to free the allocated space after the code has run.

Note The caller is responsible for allocating and freeing memory space occupied by the response string.

C/C++: Example

```
#include "iosapi.h"

int main(int argc, char** argv){
    char buf[] = "show run";
    IosServiceAPI_t iosapi;
    IosapiMessage_t msg;
    msg.size = 0;
    msg.response = NULL;
    msg.request = buf;
    getIosApi("CommonService", &iosapi);
    retcode  = iosapi.exec(&msg);
    retcode  = iosapi.config(&msg);
    return retcode;
}
```

Java: IosapiMessage

For Java, a bean object IosapiMessage is used.

```
public class IosapiMessage {
    public void setRequest(String request);
    public void setResponse(String response);
    public String getRequest();
    public String getResponse();
```

```
}
```

Java: Example

```
import com.cisco.aesop.apphosting.iosapi.*;

public static void main (String[] args){
    int status = 0;
    IosServiceAPI iosapi = IosapiFactory.getIosApi("commonservice");
    IosapiMessage msg = new IosapiMessage();
    msg.setRequest("show run");
    status = iosapi.exec(msg);
    status = iosapi.config(msg);
}
```

Perl: Example

```
use Iosapi::Iosapi;
my $request = "show run";
my $type = "CommonService";
my $val;
my $response;

$iosapi = Iosapi::Iosapi->new($type);
($val, $response) = $iosapi->exec($request);
($val, $response) = $iosapi->config($request);
```

Python: Example

```
import Iosapi
import IosapiFactory

type = "CommonService"
request = "show run"
val, iosapi = IosapiFactory.getIosApi(type)
val, response = iosapi.Exec(request)
val, response = iosapi.Config(request)
```

Configuration

Netconf over BEEP in Cisco IOS software does not support any authentication. Netconf over BEEP supports only an SASL/Anonymous profile.

For security, an ACL list can be added to the netconf configuration to allow or deny access to the netconf/BEEP connection. See "Configuration within Cisco IOS Software: Example" section on page 28.

For further information on Netconf over BEEP, see: http://www.cisco.com/en/US/products/ps6441/products_feature_guide09186a00806a5961.html.

Configuration within Cisco IOS Software: Example

```
>config t
>sasl profile JAVALIN_ANONYMOUS_SASL
>mechanism anonymous
>ip access-list standard 1
>permit host 10.1.10.2
>netconf beep listener 12345 acl 1 sasl JAVALIN_ANONYMOUS_SASL
```

```
>netconf max-sessions 16
```

Configuration within Cisco AXP Service Module: Example

```
>config t
>netconf beep initiator 1.100.50.1 12345
>netconf max-sessions 16
```

Where `1.100.50.1` is the Cisco IOS router's IP address.

Port `12345` is the port that is matches both Cisco IOS software and Cisco AXP. The number of `12345` is only an example.

Embedded Event Manager API

Package Name: axp-eemapi.<platform>.<version>.pkg

An example package name is axp-eemapi.nme.1.0.1.pkg

The Embedded Event Manager (EEM) supports Cisco IOS event notification and enables a wide range of events to be reported from Cisco IOS software to your Cisco AXP application.

The Cisco AXP Embedded Event Manager API supports the following Cisco IOS images:

- IP-Voice
- Adv-Security
- Adv-Enterprise

To register events for your Cisco AXP application, use *one* of the following two methods:

1. Use an event configuration file during packaging. This is the recommended method. See the "Registering using an Event Configuration File" section on page 29.
2. Register the event using the Cisco AXP CLI for the application. See "Registering an Event Using the Cisco AXP Service Module" in the *Cisco AXP User Guide*.

Configuration for Cisco IOS EEM is automatically performed during boot up. Configuration includes generating an EEM policy file for each event, transferring the file over Cisco IOS and configuring the EEM.

When the requested event occurs, the EEM triggers an EEM policy, which is usually a Tcl script. Execution of the event policy causes event information to be delivered to the EEM event dispatcher on the Cisco AXP Host. The dispatcher dispatches the event to the application that registered the event.

For further information on the EEM, see the following documents:

- Embedded Event Manager Overview.
- Writing Embedded Event Manager Policies Using Tcl.
- Writing Embedded Event Manager Policies Using the Cisco IOS CLI.

To notify an event using an API call, see the "EEM APIs" section on page 31.

Registering using an Event Configuration File

Specify events in an event configuration file eem_config.xml. Package event configuration files with the application and install the application on the blade. When the EEM-specific startup routine is executed during reboot, the corresponding event policy files are set and the Cisco IOS EEM is configured. The events become active.

For an example of an event configuration file and its corresponding DTD, see Event Configuration File: Example, page 30 and Event Configuration File DTD: Example, page 30. Note that the name attribute of the event element must be unique within the file.

When entering a parameter string, use double quotes for the whole string and use single quotes for substrings. For example,

```
<event name = "myeventname" type = "syslog" parameter = "pattern 'ethernet' " />
```

Note While packaging an application using the EEM API add-on package, the event configuration file eem_config.xml must be placed in directory ~/usr/eemapi.

Event Configuration File: Example

```
<Events >
    <event name=”myCLIEvent” type=”timer”  parameter = “cron name … “ />
    <event name=”myIntEvent” type=”interface”   parameter= “name Ethernet0/0  …….” />
    <event name=”myiosevent”  type=”ios_config” / >
<\Events>
```

Event Configuration File DTD: Example

```
    <!ELEMENT events (event+) >
    <!ELEMENT event EMPTY>
    <!ATTLIST event name CDATA #required>
    <!ATTLIST event parameter CDATA >
    <!ATTLIST event type ( appl | cli | counter | interface | ioswdsysmon |
        none | oir | snmp | syslog |  timer | timer_subscriber |
        ios_config)  #required>
```

The event type attribute values consist of a list of types that map to event register keywords in EEM. In addition, there are other Cisco IOS events that are not provided from EEM, such as the event ios_config.

Event Types and Command Extensions

The event types and corresponding command extensions are shown in Table 8.

Note The event type ios_config is a Cisco IOS software event type and has no parameter value.

Note The list of types in Table 8 may be incomplete: for a complete description of types, see the “EEM Policy Tcl Command Extension Reference” section of Writing Embedded Event Manager Policies Using Tcl.

Table 8 Event Types and Command Extensions

Event Type	Command Extension
appl	event_register_appl
cli	event_register_cli
counter	event_register_counter
interface	event_register_interface
ioswdsysmon	event_register_ioswdsysmon

Table 8 Event Types and Command Extensions

Event Type	Command Extension
none	event_register_none
oir	event_register_oir
snmp	event_register_snmp
syslog	event_register_syslog
timer	event_register_timer
timer_subscriber	event_register_timer_subscriber
ios_config (IOS event)	[no value]

EEM APIs

Event Notification

To notify an event using an API call from your application, use a function of the event handler to receive notification of an event. Your application must also have dependency set up upon the eemapi package (for example, package `axp-eemapi.nme.1.0.1.pkg`). Setup this dependency during packaging of your application. For further information on packaging, see the "Packaging the Application" section on page 49.

The following sections show the EEM APIs for Java, C, and Python.

Table 9 Files for EEM APIs

Language	Library/Modules/Header File
Java	eemapi.jar
C/C++	eemapi.h libeemapi.so
Python	eemapi.py

The Java API for EEM is bundled in file eemapi.jar, which contains all the interface APIs to support the receipt of EEM events.

Java

```
package com.cisco.aesop.apphosting.eemapi

EventManager.java

int Register (EventHandler eventHandler, String eventTypes)
```

Starts an event listener thread to receive EEM events.

Parameters:

`eventHandler` : object of a subclass of the EventHandler class

`eventTypes` : string of event types

Returns:

1: successful

< 0: failed

```
int Deregister()
```

Stop receiving events

Returns:

1: successful

< 0: failed

Example:

```
//get the single instance of EventManager.
EventManager em = EventManager.getInstance();
int regOk = em.Register(eventHandler, "cli interface");
int result = em.Deregister();
```

The C APIs for EEM are bundled in lib eemapi.so under directory /lib in the guest OS. You must include the header file eemapi.h in your application.

C

`eemapi.h`

```
EEMRegister (void * eventhandlerFP, char *eventTypes)
```

Start an event listener thread for receiving EEM events.

Parameters:

`eventHandlerFP` – function pointer to the application's event handler

`eventTypes` – char string of event types

Returns:

1: successful

< 0: failed

```
int EEMDeregister()
```

Stop receiving events

Returns:

1: successful

< 0: failed

Example:

```
    int result = EEMRegister( &eventHandler, "cli interface");
```

The Python API for EEM is bundled in eemapi.py under directory /lib/python2.3/eemapi in the Cisco AXP guest OS.

Python

```
eemapi.h

EEMRegister (void * eventhandlerFP, char *eventTypes)
```

Start an event listener thread for receiving EEM event.

Parameters:

`eventHandlerFP` – function pointer to the application's event handler

`eventTypes` – char string of event types

Returns:

1: successful

< 0: failed

```
int EEMDeregister()
```

Stop receiving events

Returns:

1: successful

< 0: failed

Example:

```
    int result = EEMRegister( &eventHandler, "cli interface");
```

Event Registration

Use an event registering function or method to handle events in your application.
Event handling functions and methods are described below for C, Java, and Python. Their arguments are described in Table 10.

For C, an event handler is a callback function defined in your application.

```
Void AppEventHandler (char *eventName, char *eventType, char *eventInfo)
```

For Java, an event handler is a class defined in your application, subclassed from the EventHandler class. The event handler class must overwrite the EventHandler class's callback function to receive events.

```
public class EventHandler {
    public void callback (String eventName, String eventType, String eventInfo);
}
```

For Python, an event handler is a class defined in your application, subclassed from the EventHandler class. The event handler class must override the ReceiveEvent function to receive events.

```
class EventHandler :
    def   ReceiveEvent (self, eventName, eventType, eventInfo);
```

Table 10 Application Event Handler Arguments

Parameter	Description
`eventName`	The "name" attribute of the event element in the event configuration file or the event-name specified through the CLI (see "Registering an Event using the Cisco AXP Service Module" in the *Cisco AXP User Guide*.
`eventType`	Event Type as shown in Table 8.
`eventInfo`	Result string event_reqinfo from Cisco IOS EEM. The string is formatted and specific to an event. See the "event_reqinfo" section in Writing Embedded Event Manager Policies Using Tcl.

Remote Serial Device

Package Name: axp-vserial.<platform>.<version>.pkg

The virtual serial driver package enables an application running on the application service module to access external Cisco IOS serial devices connected to the serial port of a Cisco IOS router. This allows applications to support, for example, peripherals that connect to serial ports such as GPS locators.

The Cisco IOS router's auxiliary serial ports are virtualized and appear in the Cisco AXP OS as local devices. External devices connected to the Cisco IOS router appear as standard local devices such as "/dev/tty1" or "/dev/tty2" to Linux applications hosted on application service modules. Third party applications can therefore control external peripherals attached to the router's auxiliary serial port without special knowledge of the location of the devices. Applications can open/close/read/write to the peripherals using standard Linux System calls such as: open("/dev/vtty1"), and read( ).

A reverse telnet session is established for either a line interface or a serial interface (provided that the serial interface is configured in async mode). This telnet session enables serial data transfer.

You must add axp-vserial.<platform>.<version>.pkg as a dependency when packaging your application.

Developing an Application Using Serial Device: Summary

1. Develop an application to access a serial device using standard Linux System calls such as open ( ), and read( ).

Note If a device name is hard coded in the application (such as "modem" or "gps") make sure to note the device name. This device name is used when the application binds the serial device using the CLI server.

```
int main(int argc, char *argv[]) {
    char *dev;
    int fd;
    if (argc > 1)
        dev = strdup(argv[1]);
    else
        dev = strdup("/dev/modem");
    fd = open (dev, O_RDWR | O_NONBLOCK | O_NOCTTY | O_NDELAY);
```

2. Package the application using the Bundling Tool, page 97 and include a dependency on axp-vserial.<platform>.<version>.pkg.
3. Install the application and infrastructure add-on packages onto the application service module. Make sure you either install application and packages together or install the packages before installing the application.
4. Connect the serial device to the router's serial port and configure the device on the router side using the Cisco IOS CLI. A summary of these configuration steps is shown below in Summary configuration steps from the router CLI, page 36. For more detailed configuration information, see "Remote Serial Device Configuration" in the *Cisco AXP User Guide*.

Summary configuration steps from the router CLI

```
interface serial0/0/0
physical-layer async

line 0/0/0
no exec
transport input telnet
speed 115200
```

When Serial0/0/0 interface is configured in Async mode using physical-layer async, a line 0/0/0 interface is automatically created.

Note For the line interface, ensure that **no exec** is configured.

After configuring the serial interface do a `show line` command and make sure the line associated with this serial interface has the name of the serial interface in the "int" column. Here is a sample output:

```
Tty       Line  Typ Tx/Rx      A  Modem  Roty  AccO  AccI  Uses  Noise  Overruns  Int
*   0/0/0   2   TTY 4800/4800  -  -      -     -     -     114   1      0/0       Se0/0/0
```

5. You also need to configure SASL on the Host-Router.

   Enter the following command on the Host-Router to create a SASL profile.

   ```
   sasl profile javalin
   mechanism anonymous
   ```

6. Configure netconf on both the Host-Router and the Service-Module for the serial device CLI to work. Example:

   Enter the following command on the Host-Router.

   ```
   netconf max-sessions <4-16>
   netconf beep listener <port to listen on> sasl <SASL profile to use for this session>
   ```

   Enter the following command on the service-module interface.

   ```
   netconf beep initiator <IP address of destination> <port>
   netconf max-sessions <4-16>
   ```

7. On the Service-Module Interface:

   ```
   show device serial
   ```

   This shows the device name assigned to the Cisco IOS serial interface.

   Example output:

   ```
   Device Name        TTY No.  Line No.  Line Type  Intf       Name Assigned To
   vaux                1          1        AUX       -          -       -
   ```

```
vtty000             0/0/0     2       TTY      Se0/0/0    -       -
```

If the command **show device serial** does not result in any output, see Troubleshooting: show device serial, page 37.

8. On the application service module, in the application sub-command config mode:

   ```
   bind serial <device name defined in host> <device name for the hosting environment>
   ```

   This binds the device to the application virtual instance.

   <device name for the hosting environment> is an optional device name.

   If an optional device name is specified, the name is created as an entry in the application's virtual instance /dev directory.

   If an optional device name is not specified, an actual device name such as vtty000 is created in the application's virtual instance /dev directory.

   Using the optional device name is useful in the example if the device name is hard coded in the application such as in step 1 above. If you create an application for GPS using “/dev/gps” as the device name you can bind the serial interface using:

   ```
   bind serial <device name defined in host> gps
   ```

   A “/dev/gps” entry is created in the application's virtual instance. The application can access the GPS device connected to the host router.

Your application is now able to control an external device (on the Host-Router) as if it was a local serial port.

For an example of an application accessing a serial device, see Remote Serial Device: Example, page 110.

Troubleshooting: show device serial

If the command **show device serial** does not result in any output being displayed, you can check that the NETCONF over BEEP has been properly set up in the router and the service module.

On the router you can run one of the following two debug commands:

- **show processes | include beep**
- **show processes | include NETCONF**

Either command returns a process if NETCONF over BEEP is configured correctly.

Perl

Package Name: axp-perl-5.8.8.<platform>.<version>.pkg; for example, axp-perl-5.8.8.nme.1.0.1.pkg

Perl is supported as an infrastructure add-on package. Version 5.8.8 is stable. The package is built with multiple threads support and uses ithreads (one Perl Interpreter per thread).

Development System Requirements

Applications must be developed in a Linux environment. The minimum system requirements are:

- Intel Pentium 4 CPU 1.8 GHz, 40 GB Hard Disk, and 512 MB Memory

- C Compiler: GNU C Compiler (gcc) Version 3.4.3
- Linux distribution such as Fedora Core 4, or above

Prerequisites

See the *Cisco AXP User Guide* for information on the following tasks before you install the Cisco AXP SDK.

Step 1 Install the router and application service module. See *Cisco Network Modules Hardware Installation Guide* and *Application eXtension Platform Enhanced Network Modules*.

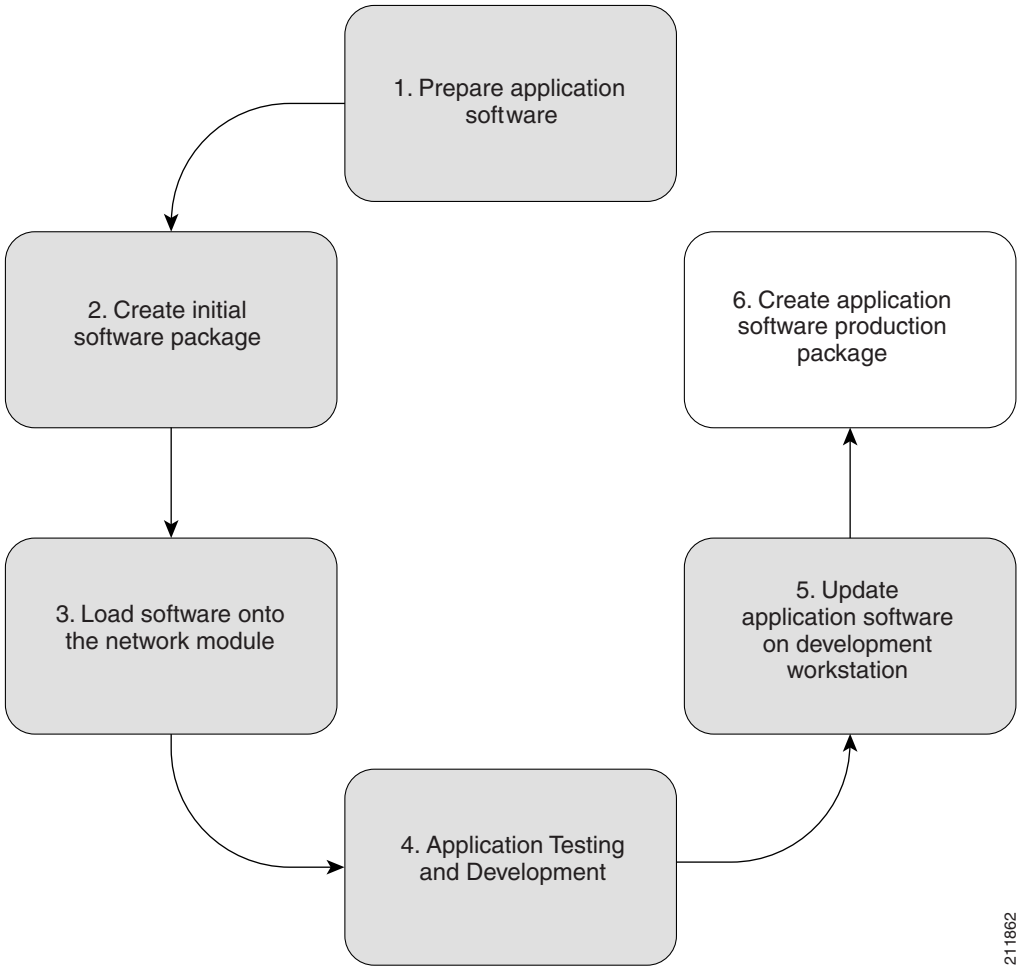
Step 2 Download the Cisco IOS software image, version 12.4(15)T3, and configure the router. See the "Software Requirements" section in the *Cisco AXP User Guide* for full details.

Step 3 After you boot up the system, configure the service module interface. See the "Configuring the Cisco AXP Service Module Interface" section in the *Cisco AXP User Guide*. Cisco AXP is preloaded on the service module.

Development Flow

This section is an overview of the main phases of application development shown in Figure 7.

Figure 7 Application Development Flow



The steps below explain the stages in the application development flow.

Step 1 **Prepare Application Software.**

a. Set up the workstation.

Use a PC with a Linux operating system, including software components that closely match those intended for the final service module runtime environment. See the "Writing the Application" section on page 47.

b. Download the Cisco AXP SDK from the Cisco Download Software Area.

If Cisco AXP software is already loaded on the Cisco AXP service module, download the SDK with the same version number as the Cisco AXP software.

c. Install the Cisco AXP SDK and prepare the workspace.

Prepare the application software to run in a Linux environment, remembering to keep the directories containing the application software separate from the workstation Linux environment. Isolate the application software in a simulated root directory; for example, /opt/. The software beneath this root directory represents the same setup as the application software when it is loaded into the virtual

environment on the application service module. The simulated root directory contains all application software and any required libraries. There is no need to include libraries or system add-on packages in the simulated root directory that are already included in the embedded guest OS.

Convert an absolute path in each symbolic link to a relative path. The symbolic links should refer exclusively to files underneath the simulated root directory.

Isolate any endpoint specific code such as license installation, site/networking configuration into a postinstall shell script. We recommend that core software is self-contained and does not have any external dependencies.

Step 2 Create the Initial Software Package.

The initial software package is used as the starting point in development and is not fully functional at this stage. Creating the initial package helps you to find any missing dependencies and customize your software in preparation for loading the software onto the Cisco AXP service module.

a. Add or change required software in the workstation's sync repository, such as executable files and scripts. The sync repository contains binary files, not source files.

b. Enable access to the Linux shell.

   Accessing the Linux shell helps you to debug the application software after the software is loaded onto the Cisco AXP service module. See the "Console Access using a Postinstall Script" section on page 14.

c. Define the dependencies of the application software on the required Cisco infrastructure add-on packages.

d. Bundle the application software with the Cisco infrastructure packages that the application depends on.

   For example, see the "Running the Packaging Tool" section on page 51.

Step 3 Load Software onto the Service Module.

a. Load the application software onto the Cisco AXP service module. For example, by using the **rsync** utility, which synchronizes files between two environments.

   For further information on rsync, see the "Rsync" section on page 105.

b. Confirm that you have access to the Linux shell.

c. Confirm that the virtual environment contains files linked in from Cisco infrastructure packages that are required by application software, e.g CLI plug-ins, OSGi™, and Tomcat files.

d. Extract any required additional files/scripts for your application using the "RPM File Extraction" section on page 44.

Step 4 Application Testing and Development.

a. Upload, to the service module, using SFTP or **rsync**, any missing application components and/or libraries.

   Iterative development can take place on the service module if required. However, you must remember to synchronize the changed software back to the workstation environment during step 5.

b. Integrate the application into the embedded virtual environment. Consider end user interface issues, including application configuration and diagnostics. If you want to extend the CLI on the development service module, use the "CLI Plug-in Distribution Service" section on page 25.

c. Test your application's functionality and stability.

   (For further information on debugging tools, see the "Troubleshooting Your Application" section on page 86, and the "Application Development Package" section on page 24.)

d. Update and test the application startup script(s).

e. Update and test the application postinstall script.

Step 5 **Update Application Software on Development Workstation.**

Copy back any application software that you have changed on the service module to the development workstation. For example, use the **rsync** CLI command.

Avoid copying infrastructure and embedded guest OS components into the application software repository. This is for space saving reasons: duplicated files unnecessarily increase the storage space taken up by the application when the application is deployed on the Cisco service module.

Step 6 **Create Application Software Production Package.**

a. On the workstation, create a new package for the application software.

b. Disable access to the Linux shell if it will not be required by the user.

c. Specify infrastructure dependencies and the postinstall script.

d. Delete any unwanted temporary files before packaging.

e. Run the library dependency checker tool to verify all libraries required by the application are in the package. See the "Library Dependency Checker" section on page 42.

f. Bundle the application software, including any required Cisco infrastructure add-on packages.

g. Install and test the packaged software.

Cisco AXP SDK

Cisco AXP provides a software development kit (SDK) for developers to use in a Linux development environment. The SDK tools and installation are explained in the following sections:

Note To return to the Table of Contents, click here.

Packaging and Bundling Tools

Packaging Tool

Use packaging tool pkg_build.sh to create and sign the application package. This tool requires that the application third party has development authorization from Cisco and access to private keys that will be used in signing application. The tool has two interfaces: command arguments, and an interactive CLI. For further information, see Bundling Tool, page 97.

Bundling Tool

Use bundling tool pkg_bundle.sh to compile multiple packages into a single bundle. The tool also compiles multiple packages into a single bundle. For further information, see Bundling Tool, page 97.

Other useful tools related to packaging are explained in the following sections:

Library Dependency Checker

The Library Dependency Checker tool (pkg_check.sh) allows you to check software dependencies before installing your application on the Cisco AXP service module.

The Library Dependency Checker tool looks for dependencies of packaged binary files in the default /lib directory. (The root directory is not checked.) The checker tool lists any binaries with unsatisfied dependencies and any corresponding missing libraries. A library in a directory such as "/opt/lib" is ignored unless it is added using the --lib-path argument.

Library Dependency Checker: Example

```
pkg_check.sh --project-dir '/xyz-src' --check-pkg AppXYZ-1.0.pkg --lib-path
'/usr/local/lib:/opt/app/lib' DependencyPkg1.pkg DependencyPkg2.pkg
```

Library Dependency Checker Arguments

Table 11 **Library Dependency Checker Arguments**

Argument	Name Associated with Argument	Description
--help	(No argument)	Version and usage information
--project-dir	Project directory	Project directory used as a temporary storage location for extracted package files.
`--check-pkg`	Package to check (optional) **or** Software root directory (optional)	Package to check (for dependencies). Must have a corresponding `.prt1` payload file stored in the same directory. Package name has `.pkg` suffix. **or** Software root directory: contains application files. **Note** Either the package **or** the software root directory must be supplied for the --check-pkg argument.
`--lib-path`	Extra path locations	The --lib-path argument allows library search path locations to be specified in addition to the default library path "`/lib`". The library path locations must be separated by a colon symbol. For example, "`/usr/lib:/usr/local/lib`". **Note** You also need to add any required library locations to LD_LIBRARY_PATH before running your application.

Table 11 **Library Dependency Checker Arguments** ***(continued)***

Argument	Name Associated with Argument	Description
--core-sys	Core System Package	Cisco AXP core system package (full path). Application dependencies are checked against the core system package. For example, "`axp-k9.nme.1.0.1.pkg`"
<list of package names>	Dependency packages	Each dependency package in the list contains files used to satisfy binary dependencies. Please note that subsystems must be specified in this list.

Library Dependencies Checker: Limitations

The Library Dependency Checker tool cannot detect the following:

- Missing dynamically loaded plugin libraries: because binaries will not contain linkage information for such libraries.
- Binaries that do not have **exec** permissions.

Package Information

The Package Information tool (pkg_info.sh) displays information about a package. The tool is located in the tools directory of the SDK.

An example of useful information that can be obtained is the SSID of a package, which can then be used to add a dependency to your application during packaging.

The SSID is specified in the deps parameter of the Packaging Tool.

Output from pkg_info.sh is placed in file: `<your project directory>/tmp/core.log`.

Example

```
pkg_info.sh sanity_test.pkg


Subsystem ID: d984142e-4dfb-4a28-bf42-60d278b7e798
Name: sanity_test
Version: 1.0
Type: plug-in
Description: Sanity
Content:
  /hello.txt  (file, not-link, data)
  /xterm  (file, not-link, executable)
  /sbin/.app_sig  (file, not-link, data)
```

Note Package information tool pkg_info.sh can only give information about one package at a time.

Note For some directories, the packaging tool displays the subsystems in that package in addition to the subsystem SSID's. The directories which have their subsystems displayed are: <SDK Directory>/pkg and <Project Directory>/pkg

RPM File Extraction

While your application is running you may discover that it stops running on the service module. To investigate this you can manually start up your application. See the "Manual Startup" section on page 14.

Start your program by entering <script name> start (a command, for example, in the startup script in /etc/rc.d/init.d).

If an error message appears that says files are missing, and a message also says that certain RPM files are required, you need to identify the RPM files to be installed so that your application can run successfully.

After you identify missing files, you can add each missing RPM file with the help of these utilities:

- Standard RPM Utilities, page 44
- RPM File Extractor Tool, page 45

Standard RPM Utilities

To add additional software after the third party's application is installed, use standard RPM utilities such as rpm and rpm2cpio. You will need access to the guest OS shell of the application on the service module.

To extract the RPM files, perform the following steps.

1. Run the **rpm2cpio** utility on the development machine.

2. Upload files from the development machine to an FTP server.

3. Download from the FTP server to the client (service module) using curl utility (see http://curl.haxx.se/docs/manpage.html). The curl utility downloads files from HTTP/FTP servers.

To install an RPM file as part of your application, perform the following steps. For more information, the "RPM File Extractor Tool" section on page 115.

Step 1 Extract the RPM file from the development environment.

To extract an RPM file to the current directory:

```
rpm2cpio /tmp/myapp.rpm | cpio -ivd
```

Step 2 Examine any install/uninstall scripts and dependencies.

To examine any pre or postinstall/uninstall scripts that the RPM file contains, use the RPM file extractor tool as follows:

```
rpm -qp --scripts /tmp/myapp.rpm
```

To examine any dependencies (libraries, or other RPM files) that the RPM requires, use the RPM file extractor tool as follows:

```
rpm -qp -R /tmp/myapp.rpm
```

See the "RPM File Extractor Tool" section on page 115.

Step 3 Examine the scripts and fit what you require into the Cisco AXP model.

RPM File Extractor Tool

The RPM File Extractor Tool (tools/rpm_extractor.sh) is part of the Cisco AXP SDK. The tool quickly extracts all the RPM files into the project source root directory. From the root directory, you can examine any dependencies the RPM files require, and view any preinstall, postinstall, or uninstall scripts that the RPMs contain.

For more information, see the "RPM File Extractor Tool" section on page 115.

CLI Plug-in Utility Tools and APIs

Cisco AXP provides a mechanism for CLI applications to be integrated into the Cisco AXP CLI environment.

A set of tools are available for you to use during development to validate, process and package the CLI plug-in along with your main application.

Value-added Service APIs

Value-added service APIs allow application programs to access, manage, and augment the existing features of Cisco IOS software. The Cisco AXP SDK includes libraries, APIs, and associated header files. The Cisco AXP SDK allows you to compile applications and link applications in a development environment.

To obtain a value-added service API, install the package that you require from the "Infrastructure Add-on Packages" section on page 23 and package your application with a dependency upon the add-on package.

To allow your application to use the service, install both the infrastructure add-on package and your application package onto the service module.

Installing the SDK

The SDK includes the directories and files, see Table 12.

Table 12 Cisco AXP SDK Directories and Files

Directory/File	Description
include	CLI distribution header file.
jar	CLI distribution jar file.
lib	pkg, bin, and scripts subdirectories.
perl	Perl module files.
python2.3	Python module files.
tools	Application packaging and bundling tools.

To install the Cisco AXP SDK:

Step 1 Download the SDK file **axp-sdk.<package>.tar.gz** from the Cisco Download Software Area (or Cisco File Exchange).

Step 2 Create a directory for the SDK:

```
mkdir –p /source/sdk
```

Step 3 Extract the contents of the SDK file into the SDK directory:

```
tar xvzf axp-sdk.<package>.tar.gz –C /source/sdk
```

The SDK is now installed. If the `<package>` in the file `axp-sdk.<package>.tar.gz` is “1.0.1”, the SDK directory name includes “1.0.1”.

Developing an Application

Developing an application is explained in the following sections.

Also see the “CLI Plug-in Applications” section on page 56.

Writing, Testing, and Packaging an Application

In general, the first step in the development process is to write your application along with startup/shutdown scripts in your Linux development environment. After testing that your application works properly, copy your application and the relevant scripts, extracted RPM files and any other required files to your build directory. You then create a package directory, and run the Cisco AXP tool that packages your application so that it can be installed onto the Cisco AXP system.

Certificates

There are several steps to be followed when packaging your application before installing the application on Cisco AXP. One required step is to create a signed certificate for your application. Any application to be installed on Cisco AXP must have a signed certificate, a private key, and a development authorization certificate which is provided by Cisco. For more information see the “Creating Certificates” section on page 50. Additionally, consider allowing access to the OS shell of your application and protecting files.

Shell Access and Security

Each application installed onto Cisco AXP has its own virtual instance (also see the “Cisco Application eXtension Platform Overview” on page 2). Inside the instance the application has its own shell and environment. Access the guest OS shell for debugging, modifying/viewing specific files for your program, and importing other applications and tools. However, allowing the user to access the shell increases the security risk in your application. Access to the guest OS shell access is described in the “Packaging the Application” section on page 49.

If you decide to allow someone to have shell access to your application environment, there may be certain files that you want to protect from modification and/or deletion. To protect files, see the “Creating a File of Protected Files” section on page 50. If any of these listed files are modified and/or deleted, then when the virtual instance for your application next tries to restart, the modifications/deletions will be reported and the virtual instance will not start. This prevents your application from running. A message is written to the system log listing the name of the modified/deleted file.

Writing the Application

Your application should be written and tested in the Linux development environment before attempting to install the application on Cisco AXP.

Cisco AXP is based on Cisco Linux. Libraries are listed in the appendix. You must import all additional code features for your application into Cisco AXP. This code may include: start up and kill scripts, softlinks to these scripts, and any additionally required RPMs.

We recommend that the health status is coded in the application in order for the **show state cli** command to show the application's health status.

Note The size of the /tmp directory in the virtual instance is limited to 16MB.

For your application to have console access, you need to either have a post-install script or your application must depend upon debug package axp-app-dev.<platform>.<version>.pkg.
For more details of the post-install script and the debug package, see the "Obtaining Console Access for the Application" section on page 14.

C Applications

When developing native C/C++ software targeted to run on Cisco AXP, we recommend that you use the following components:

- Compiler GCC/G++ 3.4.3
- Library glibc: 2.3.5
- Library libstdc++: 6.0.3

Fedora 6 C/C++ applications must be linked with one of the two following linker options:

- `-Xlinker -hash-style=sysv`
- `-Xlinker -hash-style=both`

Hello World Application: Example

The following example code shows a simple "Hello World" application.

The application is a Bash script that reports on its health status every five seconds by writing to a file called fancy_output.

Step 1 Create a directory in which to save a bash script using the following command:

```
mkdir /source/helloworldapp
```

Step 2 Create a script called helloworld.sh containing the following lines of code:

```
#!/bin/bash
#provide health status
/bin/app_status_notifier helloworld INITIALIZING
# remove old log file
rm -rf fancy_output
#provide health status
/bin/app_status_notifier helloworld ALIVE
while [ 1 ]; do
    echo "Hello world!" >> fancy_output
    sleep 5
done
```

Step 3 Save helloworld.sh into directory /source/helloworldapp.

Step 4 (Optional) There are two ways to view the file fancy_output without accessing the guest OS shell.

Select one of the following two steps:

- Write to the fancy_output file in directory /var/log/. For example:

```
echo "Hello world!" >> /var/log/fancy_output
```

- Create a softlink from file fancy_output's directory to a file that you will create in directory /var/log/. For example, add the following lines to your postinstall script:

```
touch /var/log/fancy_output
ln -s /var/log/fancy_output /helloworld/fancy_output
```

File fancy_output is written to directory /helloworld. Cisco AXP does not allow you to create a softlink from the /var/log directory to your file because this could potentially lead to a security breach.

Step 5 (Optional) View your file from the guest CLI:

```
show log name <your file name>
```

Automatic or Manual Application Startup

The application starts up in one of the following two ways:

- Automatic Startup, page 48
- Manual Startup, page 48

Automatic Startup

For automatic startup on bootup of Cisco AXP, create a startup script, normally created in /etc/rc.d/init.d and reference it with a softlink placed in your /etc/rc.d/rc3.d directory. Remember that your startup softlink should be formatted as S<priority><app-name> where S means startup, priority is 0–99, and app-name is the name you want to call your startup script.

Create a kill softlink that references your startup script, and put the script in your /etc/rc.d/rc6.d directory. Recall that your kill softlink should be in the format:

K<priority><app-name> where, K means kill, priority is 0–99, and app-name is the name you want to call your startup script.

Package your application with these softlinks and references to the startup script and import them into Cisco AXP.

Manual Startup

For manual startup, first configure your system for shell access. For more information, see the "Obtaining Console Access for the Application" section on page 14.

Packaging the Application

This section consists of the following sections:

Note To return to the Table of Contents, click here.

Creating a Common Root Directory

Create a common root directory such as /source/helloworldapp/build.

This directory is the common root directory from which the packages are created and installed on Cisco AXP.

Copying Files

- Copy your start up and kill soft links located in /etc/rc.d/rc3.d and /etc/rc.d/rc6.d with their directory structure intact, into the common root directory.
- Copy your referenced script /etc/rc.d/init.d into the common root directory.
- Copy all other files and links required for your application into the common root directory.

Hello World Build Directory: Example

In this example, the common root directory is /source/helloworldapp/build.

The subdirectories beneath the common root directory are:

```
./bin:
post-install.sh

./etc/rc.d/init.d:
helloworld

./etc/rc.d/rc3.d:
S99helloworld -> ../init.d/helloworld

./etc/rc.d/rc6.d:
K99helloworld -> ../init.d/helloworld

./helloworldapp:
hello_world.sh

./sbin:
```

Creating a File of Protected Files

If you decide to allow a user to access the guest OS Shell, you can optionally protect files from being changed. This may include the startup script, kill script, helloworld, and process script helloworld.sh. This is done by creating a file that contains a list of protected files.

Create file called protect.txt that holds a list of protected files. Include any necessary file path before each file name. **Note:** There should be only one file in each line of the protect.txt file.

During packaging, a checksum is run on each of the files in protect.txt, and the results are stored along with the package. When the virtual instance starts, protected files are verified against the checksums to ensure the integrity of the files.

Protected File List: Example

```
/etc/rc.d/init.d/helloworld
/helloworldapp/helloworld.sh
```

Creating Certificates

Certificates and a private key file are required during packaging. This developer certificate is an X.509 certificate, signed by a trusted certificate authority, for example VeriSign. Please refer to your certificate authority for instructions on how to generate a certificate request and obtain a signed certificate. We recommend that you use a key length of 1024 bits or greater for certificate keys, using RSA public-key encryption.

Step 1 Select one of the following two options:

- Obtain an X.509 certificate from a certificate authority or
- Create a self-signed X.509 certificate. Perform the following steps.
  - Generate a self-signed X.509 certificate with the OpenSSL tool. Create an RSA key pair. The resulting private key file will contain a private key:

    openssl genrsa -out private.key 1024
  - Create a self-signed X.509 certificate valid for one year:

    openssl req -new -x509 -days 365 -key private.key -out dev_certificate.sig

Note Note: The developer certificate must be named dev_certificate.sig.

Step 2 Send the developer certificate to us.

Step 3 If the developer certificate is approved, we will return the authorization certificate: dev_authorization.sig.

Step 4 If the directory does not exist, create directory /source/certs

Step 5 Move the authorization certificate (including the private key file) to directory /source/certs.

Creating a Package Directory for Project

Create a package directory for your project. This directory includes packaged files to be installed on Cisco AXP. For example: `mkdir /source/helloworldapp/package`.

Running the Packaging Tool

Note The packaging tool expects bash scripts to be version 3.0 or above.

Step 1 Run the SDK packaging tool pkg_build.sh located in directory axp-sdk.1.0.1/tools

Specify parameters to the packaging tool interactively or in batch mode (in a file).

- To specify parameters from a file, supply the file with values for the list of parameters that are found in the "Bundling Tool" section on page 97.
- To specify parameters interactively, after you enter "pkg_build.sh" you are prompted to enter each parameter separately. See the "Packaging the Hello World Application: Example" section on page 51.

You can build a package from any location in the file system.

Step 2 (Optional) Run a postinstall script.

Execute the postinstall script after installation and before the application first starts.

Specify the location of the script using a relative path with respect to the common root directory. For example, if script postinstall.sh is in directory /source/build/bin, and the common root directory is /source/build, specify the location of the script as /bin/postinstall.sh

Packaging the Hello World Application: Example

This example shows the packaging tool pkg_build.sh being used in interactive mode to package the "Hello World" application. For a description of the packaging tool modes, and parameters, see the "Bundling Tool" section on page 97. After packaging, to install the application, see the "Installing, Uninstalling, and Upgrading an Application" section on page 54.

In interactive mode, the tool prompts you to input each parameter, starting with the parameter ****project-dir**.

1. A description of the project directory parameter is displayed.
2. You are prompted for "Project Directory:"
3. Input the project directory name, with path. For example:

   ```
   /source/helloworldapp/package
   ```

   The next parameter, **** dev-cert** is displayed.

Packaging Tool: Example

Note Optional parameters such as **`** uuid`** have been left blank in this example.

```
**project-dir
Used to store output packages as well as any intermediary files, for example manifest
files. Project directory must be readable and writable.
Project Directory: /source/axp_tools/axp-sdk.1.0.1/tools/pkg_build.sh

** dev-cert
```

Development certificate will be used to verify software package integrity during installation and runtime (if the feature is enabled). The certificate must be in X.509 format. Development certificate must be authorized for development on this platform. Please contact Cisco for information on development authorization.

Development Certificate File: /source/helloworld_app/certs/dev_certificate

**** dev-auth**
Development authorization verifies that development certificate can be used for software development. Development authorization contains cryptographic signature of development certificate. Please contact Cisco if you need to obtain a development authorization file for your development certificate.

Development Authorization File: /source/helloworld_app/certs/dev_authorization

**** private-key**
Private key in this file is used to sign application package files. The public key corresponding to this private key must be stored in development certificate.

Private Key for Signing Operations: /source/helloworld_app/certs/private.key

**** name**

Name of the application to be packaged: used to identify application through the command line interface(CLI) as well as for naming package files.

Application Name: helloworld

**** version**
Application version is used when displaying application information through command line interface. It is also used for upgrade/downgrade sequencing.

Version: 1.0

**** description (optional)**
Application description is used when displaying application information through command line interface. For example it may contain a statement about application's capabilities.

Description: hello hello

**** uuid (optional)**
Unique identifier to be used with this application. The identifier can be generated using Linux 'uuidgen' utility. Automatically generated if left blank.

Unique Identifier:
Using Default Value:

**** source-dir**
Source directory containing the files to be packaged. Files must be laid out in the same as they will appear in the root file system when the files are installed in the application hosting environment.

Source Directory: /source/helloworld_app/build

**** protect (optional)**
File that contains a list of application files to be protected. All files listed in this file will be verified during system startup. No application files will be if this parameter is left blank.

Protect List File:
Using Default Value:

**** enable-license (optional)**

```
Enables application licensing: the application will not start if the license for the
application is not present.

Default Value: FALSE
Enable Application Licensing (y/n): n


** deps (optional)
Application may depend on one or more components outside of its own
package. Dependencies are specified based on application's unique
identifiers. This tool can lookup application unique identifiers
for packages in '/pkg' and '/pkg' directories.
If this option is not specified the application will have no
dependencies

Loading existing SLIM Packages in directories:
  /source/sdk/tools/../pkg
  /source/helloworldapp/package/pkg

Dependency Subsystem Identifier:

** disk-limit
disk limit option specifies the minimum disk space allocated
for the application. When disk resource falls below this limit, the
application will not be allowed to install. Minimum disk limit must be
specified in megabytes, for example 1500M.

Disk Limit: 10M

** memory-limit
Memory limit option allocates RAM space for the application.
Memory limit must be specified in megabytes, for example 256M
Memory Limit: 64M

** cpu-limit
CPU limit option specifies the minimum amount of cpu resource
allocated for the application. When the cpu resource falls below this
limit, the application will not be allowed to install.
This option is uniform across supported platforms so the same package
can be installed on any of the supported platforms of the same CPU
architecture. Please refer to developer's guide for the information on
how to calculate CPU limit

Cpu Limit: 1000

** postinstall (optional)
Postinstall script is executed after installation is complete but before
the application has been started for the first time. Postinstall script
must be packaged with the application. When entering postinstall script
location, use relative path with respect to source tree directory.

Post-install script path: bin/post-install.sh

Interactive mode complete. Resulting CLI command:

------------------
/source/axp_tools/axp-sdk.1.0.1/tools/pkg_build.sh --project-dir
'/source/helloworld_app/package' --dev-cert
'/source/helloworld_app/certs/dev_certificate.sig' --dev-auth
'/source/certs/dev_authorization.sig' --private-key '/source/certs/private.key' --name
'helloworld' --version '1.0.1' --description 'hello hello'  --source-dir
'/source/helloworld_app/build' --disk-limit '10M'  --memory-limit '64M' --cpu-limit '1000'
--postinstall 'bin/post-install.sh'
------------------
```

```
Press enter to continue...

SLIM packaging core log file saved to: /source/helloworld_app/package/tmp/core.log
Renaming helloworld.pkg -> helloworld.1.0.pkg
```

Tip You can save the above "Resulting CLI command" to use as the basis of a script file for future packaging.

Installing, Uninstalling, and Upgrading an Application

The following sections describe installing, uninstalling, and upgrading an application.

Note To return to the Table of Contents, click here.

Installing an Application

The section below explains how to install a software package/bundle.

Installation Commands

To install a bundle that includes either your application and Cisco AXP host OS, or your application with Cisco infrastructure add-on packages and Cisco AXP host OS, use the **software install clean** command. After clean installation, the startup configuration is cleared back to the factory default. This is similar to the effect of a **write erase** command.

To install a bundle that includes either your application, or your application with Cisco infrastructure add-on packages, use the **software install add** command.

Installing Package from an Ftp Directory

To install a package via an a) an ftp anonymous directory, or b) a non-anonymous ftp directory, use one of the following commands:

a. **software install add url** *url*

   for an anonymous ftp directory; for example, where *url* is ftp://<server>/helloworld.nme.1.0.1.pkg

b. **software install add url** *url* **username** *username* **password** *password*

   for a non-anonymous ftp directory (requiring a username and password); for example, where *url* is ftp://<server>/<dir>/helloworld.nme.1.0.1.pkg

Note The system reboots after installing the package.

Uninstalling an Application

Enter the following command on the service module to uninstall an application:

software uninstall

Select the subsystem to be removed from the menu displayed.

Resources allocated to the application are released and claimed by the Cisco AXP host OS.

Upgrading an Application

Upgrading an infrastructure add-on package (developed by us) and upgrading an infrastructure add-on package (developed by a third party) is supported by the Cisco AXP installer.

The upgrade may be described as “any-to-any” which means you can upgrade from any version of the application to any other version of the application. The new package must be a complete replacement of the old package with the same name. After upgrading, during system startup, the saved startup configuration is restored.

To upgrade an add-on, refer to section “Installing and Upgrading Software” in the *Cisco AXP User Guide*. The main commands for upgrading are summarized as follows:

- If the package is already downloaded onto the blade:

  software install upgrade <pkg file name>

- If downloading from a URL using ftp (with anonymous username):

  software install upgrade url <url>

- If downloading from a URL using ftp (with username and password):

  software install upgrade url <url> username <username> password <password>

Cisco AXP supports upgrading to a lower version of the application. This “downgrading” of the application refers to when the version of the application/package being installed is lower than the current application version.

Verifying Your Application is Installed and Running

To verify your helloworld application is installed, enter the following command.

Step 1 **app-service ?**

If the application is installed, the application appears in the list.

Step 2 **app-service helloworld**

If the application is running, the system brings you into the CLI of the guest OS shell.

Step 3 **show processes**

The application appears in the list of processes. For example, helloworld.

Step 4 (Optional) **show state**

Processes and their states are listed. (Provided the health status was programmed in the applications).

Example:

```
APPLICATION STATE       HEALTH
```

```
helloworld online     ALIVE
```

CLI Plug-in Applications

Cisco AXP CLI plug-ins allow you to create application-specific commands to join those commands already available via the Cisco AXP CLI.

The first five sections below tell you how to develop a CLI plug-in application.

Note To return to the Table of Contents, click here.

CLI Plug-in Distribution Service

The Cisco AXP CLI plug-in distribution service (axp-cli-plugin.<platform>.<version>.pkg) is an Infrastructure Add-on Package that provides the necessary libraries to distribute CLI plug-ins from the Cisco AXP CLI server to your application. For further details, see the "Infrastructure Add-on Packages" section on page 23.

CLI Plug-in Application Components

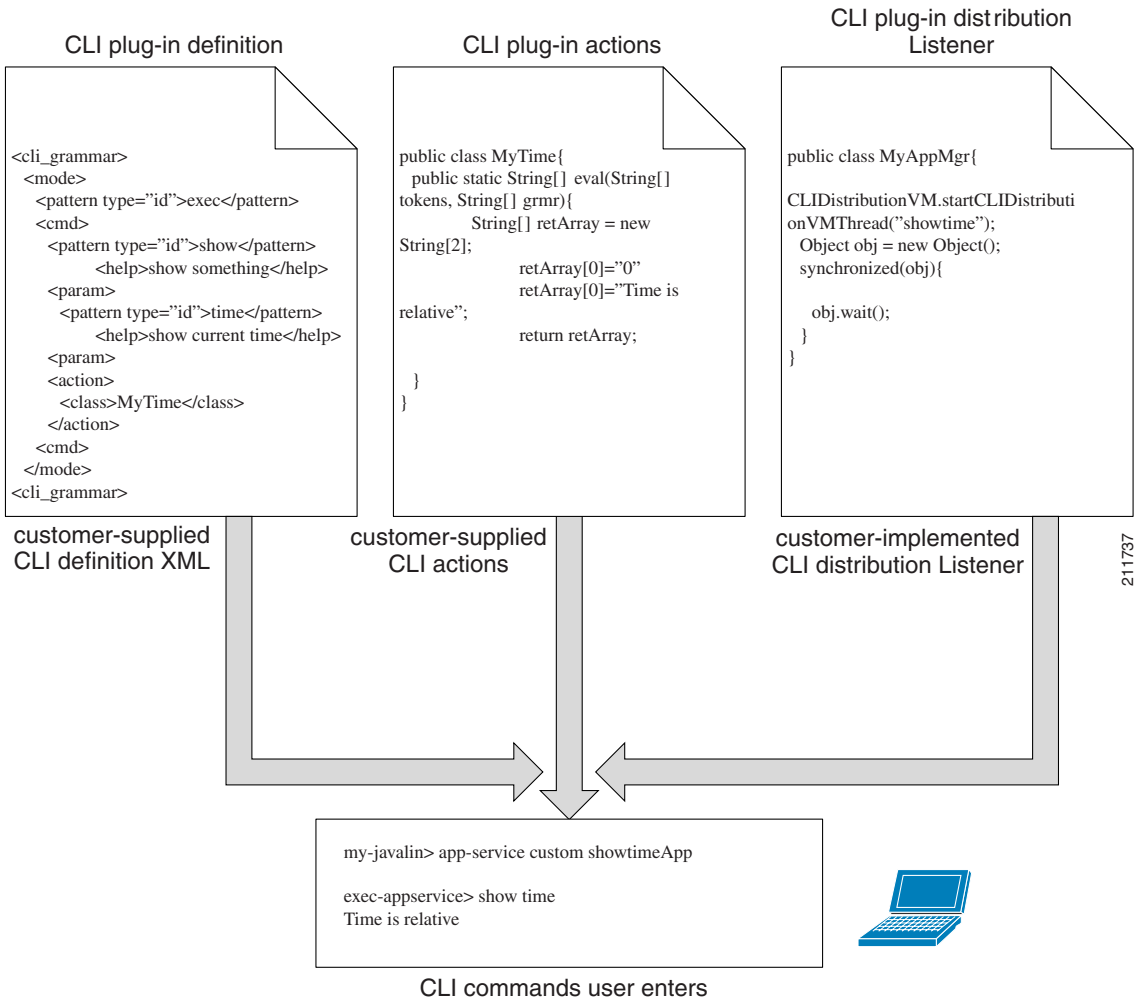
To integrate commands into Cisco AXP, the required components of a CLI plug-in are shown in Table 13.

Table 13 CLI Plug-in Application Components

CLI Plug-in Component	Description
XML definition file	Defines the syntax of the command entered by the user. The XML for the command must adhere to the Data Type Definition (DTD) shown in the "XML DTD Definition" section on page 61.
Action class	A Java class, C class, or Bash script that runs when the user enters a CLI command that is defined in the XML definition file. **Note** The CLI plug-in action must be the same language as the corresponding CLI plug-in application (distribution listener). Java: When writing an action script (class) using Java, compile the class into **<action name>.class**. C: When writing an action script (class) using C, compile the class into a shared library called **<action name>.so**. Bash: When writing an action script using a Bash Shell script, <action name>.sh, note that only an integer value may be returned indicating the exit status. There is no support for returning messages for display by the CLI console.
Application	Starts a CLI VM thread **startCLIDistributionVMThread** that listens to user commands. Only if the application is running will the thread be able to listen and respond to CLI commands entered by the user. When the user enters a CLI command defined in the XML definition file, **startCLIDistributionVMThread** invokes the corresponding action class.

Figure 8 shows these three CLI plug-in components, which are set up to process the **show time** command.

Figure 8 Cisco AXP CLI Plug-in Service



Packaging a CLI Plug-in Application

Ensure your CLI plug-in prepackaged files have been copied into the /source/showtime/build directory. For example, see the "Prepackaging the CLI Plug-in: Example" section on page 85. If the CLI plug-in's prepackaged files are in a tar file, extract them to the package directory. For example, use the following command to untar the showtime.tar file:

```
tar xvf showtime.tar -C /source/showtimeapp/build
```

To package the application, follow the steps in the "Copying Files" section on page 49.

Choose *one* of the following two packaging options:

- **axp-cli-plugin as a dependency**
  - Install axp-cli-plugin.<platform>.<version>.pkg separately onto Cisco AXP.
  - Package the application (using the "Bundling Tool" section on page 97), with a dependency of the SSID of the axp-cli-plugin.<platform>.<version>.pkg.

  - Install the application package onto Cisco AXP.

or

- **axp-cli-plugin bundled with application**
  - Package the application (using the Bundling Tool, page 97), with a dependency of the SSID of the axp-cli-plugin.<platform>.<version>.pkg.
  - Bundle the axp-cli-plugin.<platform>.<version>.pkg with the application package using the Bundling Tool, page 97.
  - Install the bundled package onto Cisco AXP.

Installing a CLI Plug-in Application

To install a CLI plug-in application, perform the following steps.

Step 1 See the "Installing, Uninstalling, and Upgrading an Application" section on page 54.

Step 2 See the "Verifying Your Application is Installed and Running" section on page 55.

Step 3 See the "Activating a CLI Plug-in Application" section on page 59.

Activating a CLI Plug-in Application

To invoke a CLI plug-in application (in EXEC mode), enter:

```
app-service <application name>
```

Activating CLI Plug-in Commands: Example

In the "Show Time Application: Example" section on page 80 a plug-in for the command **show time** was created. To invoke this CLI plug-in, enter:

```
app-service showtime
```

To display current time on the Cisco AXP service module:

```
show time
```

XML DTD File

A CLI command for a CLI plug-in written by a third-party is defined using XML. The XML must conform with the XML Document Type Definition (DTD) file. The DTD file allows you to plug a CLI plug-in application into the Cisco AXP CLI server.

An explanation of how to define a CLI command using an XML definition file, and the individual action classes invoked for the CLI command, are explained in the "CLI Plug-in Applications" section on page 56.

To understand this section, you need to know about DTD files. To find out more about DTD files or to refresh your knowledge, see http://www.w3schools.com/dtd/default.asp or another source of DTD training.

DTD Introduction

In a DTD, the declarations for elements, attributes, and entities are prefaced in the same way, with an opening angle bracket followed by an exclamation mark: <!. After the exclamation mark comes one of three keywords written in all capitals—ELEMENT, ATTLIST, or ENTITY—that specify the type of declaration.

The syntax of the ELEMENT is:

```
<!ELEMENT element-name (child1,child2,...)>
```

, where child1, child2 are child elements of element-name.

The syntax of ATTLIST is:

```
<!ATTLIST element-name attribute-name attribute-type default-value>
```

The ATTLIST keyword is followed by the name of the element `element-name` for which the attributes are being defined and the list of attributes.

Occurrence Operators

The child elements of ELEMENT are shown above as : "`(child1,child2,...`".

The element `child1` indicates there is exactly one occurrence of the `child1` element. Use the following operators to control if an element is optional and how many elements occur together.

- The question mark (?), which makes an element optional but not repeatable: the element may be used once or not at all. For example, `(child1?)`.
- The plus sign (+), which makes an element not optional and repeatable: the element must be used at least once. For example, `(child1+)`.
- The asterisk (*), which makes an element optional and repeatable: the element may be used any number of times or not at all. For example, `(child1*)`.

Element Choice

You may nest a choice of elements by placing them in parentheses and separating them by a vertical bar. For example, `(e1|e2)` indicates an occurrence of either element `e1` or element `e2`.

CLI DTD

The CLI DTD file is used to define the format of the XML that specifies a CLI command. The DTD file, which is part of the Cisco AXP SDK, is located at tools/custom_cli/cli_grammar.dtd.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!ELEMENT cli_grammar (mode)+>
<!ELEMENT mode (pattern, (cmd)*)+>
<!ELEMENT pattern (#PCDATA)>
<!ATTLIST pattern type (id|regex|help) "id">
<!ELEMENT cmd ((pattern,(help)?), ((param)*),action)>
<!ELEMENT help (#PCDATA)>
<!ELEMENT param (pattern,(help)?)+>
<!ELEMENT action ((class)?, (interrupt-timeout)?)>
<!ATTLIST action
    type (default) "default"
    paginate (yes|no) "no"
    pipehelp (yes|no) "no">
<!ELEMENT class (#PCDATA)>
<!ELEMENT interrupt-timeout (#PCDATA)>
```

XML DTD Definition

The elements and attributes of the cli_grammar DTD are explained as follows:

cli_grammar

The cli_grammar element defines the top level document type for CLI XML files. All CLI XML files must start by declaring this document type and specifying the “cli_grammar.dtd” definitions file.

```
<!ELEMENT cli_grammar (mode)+>
```

Each cli_grammar element consists of one or more `mode` elements.

```
<!DOCTYPE cli_grammar SYSTEM "cli_grammar.dtd">
<cli_grammar>
    <!--  The contents of the CLI XML file are place here -->
</cli_grammar>
```

mode

```
<!ELEMENT mode (pattern, (cmd)*)+>
```

There are one or more `mode` elements. Applications must define at least one `mode` element within the XML file. The `mode` element defines the CLI mode to which the subsequent XML elements will apply.

Each `mode` element contains one `pattern` element and zero or more `cmd` elements.

The `pattern` element has two type attribute values: `exec` and `config`, which represent the two common CLI modes.

mode: Example

```
<cli_grammar>
    <!--  Place valid mode elements here -->
    <mode>
         <pattern type="id">exec</pattern>
        ...
    </mode>
```

```
        ...
    <mode>
         <pattern type="id">config</pattern>
        ...
    </mode>

</cli_grammar>
```

pattern

```
<!ELEMENT pattern (#PCDATA)>
```

There is one `pattern` element, containing parsed character data (#PCDATA).

The parsed character data in the `pattern` element defines a name string used by the CLI for token matching. The `pattern` element is used within two different elements in the CLI XML:

1. `pattern` element within the `mode` element—defines the "CLI mode" section on page 62
2. `pattern` element within the `cmd` element—defines the "CLI Token String" section on page 62

CLI mode

In the case of a `pattern` element defined within the `mode` element, the name specified in the `pattern` element defines the name of the CLI mode to which the subsequent XML elements apply. The following example uses the `pattern` element to declare the CLI "exec" mode.

CLI mode: Example

```
<cli_grammar>
    <mode>
        <pattern>exec</pattern>
        <!--  Place additional mode elements here -->
    </mode>
</cli_grammar>
```

CLI Token String

The `pattern` element can also be declared within the `cmd` element. The `pattern` declares the token string to be matched by the CLI parser.

The following example shows the `pattern` element is used to declare the CLI token string "show". The "show" command is used in "exec" mode. The CLI server looks for an exact match of the token string defined in the `pattern` element.

CLI Token String: Example

```
<cli_grammar>
    <mode>
        <pattern>exec</pattern>
        <cmd>
            <pattern>show</pattern>
        </cmd>
        <!--  Place additional mode elements here -->
    </mode>
</cli_grammar>
```

type attribute (within pattern element)

```
<!ATTLIST pattern type (id|regex|help) "id">
```

The `type` attribute of the `pattern` element has three possible values: id, regex, and help.

The `type` attribute describes the pattern. The three supported `type` attribute values are:

1. type="id" Defines the exact keyword string.

   As "id" is the default attribute value, the following two DTD pattern element definitions are identical:

   ```
   <pattern type="id">trace</pattern>


   <pattern>trace</pattern>
   ```

2. type="regex" Defines a Perl5 regular expression. See http://perldoc.perl.org/perlre.html. Example:

   ```
   pattern type="regex">[abc]</pattern>
   ```

3. type="help" Defines the token displayed in place of a regular expression when the "?" help is entered on the command line. Extra text is defined using a `help` element. Example:

   ```
   <cli_grammar>
   <mode>
       <pattern>exec</pattern>
       <cmd>
           <pattern type="id">show</pattern>
           <pattern type="regex">[abc]</pattern>
           <!—Specifies the help WORD for display in place of the regex -->
           <pattern type="help">A-B-C</pattern>
           <help>Select A or B or C</help>
       </cmd>
   </mode>
   <!-- Place additional mode elements here -->
   </cli_grammar>
   ```

Enter **show?** to display:

```
A-B-C Select A or B or C
```

help

```
<!ELEMENT help (#PCDATA)>
```

There is one occurrence of the help element, containing parsed character data (#PCDATA).

The element help defines the position sensitive help text that will appear on the console in response the '?' entered at the command line. The help element is associated with the previous pattern element in the CLI XML. We recommend that every pattern element have an associated help pattern specified to provide meaningful online help information.

The following lines show the help that appears for the exec mode command "show user".

```
hostname> ?
show       Show running system information
hostname> show?

user       Display user information
hostname>
```

The XML supporting the above help output is:

```
<cli_grammar>
    <mode>
        <pattern type=”id”>exec</pattern>
        <prompt></prompt> <!--no prompt for exec mode-->
        <cmd>
            <pattern type=”id”>show</pattern>
            <help>Show running system information</help>
            <param>
                <pattern type=“id”>user</pattern>
            <help>Display user information</help>
            </param>
        </cmd>
    </mode>
    <!-- Place additional mode elements here -->
</cli_grammar>
```

param

```
<!ELEMENT param (pattern,(help)?)+>
```

There are one or more `param` elements, with one child `pattern` element and zero or one child `help` elements.

The `param` element defines keywords/tokens on the CLI command line *in addition to* the primary keyword. Each `cmd` element defines a complete CLI command line which will consist of the primary keyword `pattern` element optionally followed by zero or more `param` elements each specifying their own keyword `pattern` elements.

The following example has the minimum number of definitions required to construct the EXEC mode command line “turn on the lights”.(To make the example easier to understand, help elements, as well as optional elements and attlists have been removed.) Note that the primary keyword is defined by:

```
            <pattern type=”id”>turn</pattern>
```

prior to the param elements which define the remaining keywords on the command line.

```
<cli_grammar>
    <mode>
        <pattern type=”id”>exec</pattern>
        <prompt></prompt> <!--no prompt for exec mode-->
        <cmd>
            <pattern type=”id”>turn</pattern>
            <param>
                <pattern type=“id”>on</pattern>
            </param>
            <param>
                <pattern type=“id”>the</pattern>
            </param>
            <param>
                <pattern type=“id”>lights</pattern>
            </param>

            <action>
            <class>com.home.lights.exec_lights_on</class>
            </action>
        </cmd>
    </mode>
    <!-- Place additional mode elements here -->
</cli_grammar>
```

cmd

```
<!ELEMENT cmd ((pattern,(help)?), ((param)*,action)>
```

The `cmd` element contains an optional child `help` element, zero or many `param` elements, and one `action` element.

The `cmd` element is the core building block for the CLI command definition. The `cmd` element is used to construct CLI parser nodes within the CLI server. The type of CLI parser node and the node's behavior is derived from the child elements of the `cmd` element.

cmd: Example 1

Defines the CLI command "show user database" and a CLI eval( ) handler to the com.cisco.aesop.cli.commands.exec_show_user_database.class class.

The CLI server is instructed to enable pagination during the formatting of the output of the command and the CLI server is instructed to enable the use of the pipe "|" text search mechanism. Refer to the `paginate` attribute within the `action` element.

```
<cli_grammar>
    <mode>
        <pattern type=”id”>exec</pattern>
        <prompt></prompt> <!--no prompt for exec mode-->
        <cmd>
            <pattern type=”id”>show</pattern>
            <help>Show running system information</help>
            <param>
                <pattern type=“id”>user</pattern>
                <help>Display user information</help>
            </param>
            <param>
                <pattern type=“id”>database</pattern>
                <help>Display all the CLI commands</help>
            </param>
            <action paginate=”yes” pipehelp=”yes”>
                <class>com.cisco.aesop.cli.commands.exec_show_user_database.class</class>
            </action>
        </cmd>
    </mode>
    <!-- Place additional mode elements here -->
</cli_grammar>
```

cmd: Example 2

This example defines a CLI configuration command "`hostname <name>`". It defines the CLI `eval( )` handler for this CLI command to be class com.cisco.aesop.platform.conf_hostname.class.

The hostname entered at the command line is accepted as a regular expression meeting the rule defined below, and a "?" help response is provided for each keyword on the command line including the regular expression.

```
<cli_grammar>
    <mode>
        <pattern type=”id”>config</pattern>
        <prompt>(config)</prompt>
        <cmd>
            <pattern >hostname</pattern>
            <help>set the system name</help>
            <param>
                <pattern type="regex">[a-zA-Z][-a-zA-Z0-9]*[a-zA-Z0-9]/pattern>
                <pattern type="help">name</pattern>
```

```
                <help>hostname not including the domain</help>
            </param>
            <action>
                <class>com.cisco.aesop.platform.conf_hostname.class</class>
            </action>
        </cmd>
    </mode>
    <!--  Place additional mode elements here -->
</cli_grammar>
```

action

```
<!ELEMENT action ((class)?, (interrupt-timeout)?)>
```

One `action` element contains an optional `class` and optional `interrupt-timeout` element.

The element action defines the action the CLI server will take when a CLI command match is declared by CLI parser.

type attribute

```
<!ATTLIST action type (default) "default" paginate (yes|no) "no" pipehelp (yes|no) "no">
```

The `action` element supports the `type` attribute. The `type` attribute currently only supports the default attribute-type, and may be expanded in the future to support additional action element types. Therefore for now, the `type` attribute can be left out of the CLI XML action element definitions.

paginate attribute (within type element)

```
<!ATTLIST action type (default) "default" paginate (yes|no) "no" pipehelp (yes|no) "no">
```

The `action` element supports the `paginate` attribute. The `paginate` attribute instructs the CLI server to format the text output from the action class eval( ) handler method. The CLI server breaks the output text stream every 24 lines and displays the "`-- More --`" message to the user. The user can then press any key to display the next page of output.

paginate: Example

The following example shows how to create a command ("`show parser commands`") which specifies pagination of the output. The command invokes the eval( ) method in the handler class com.cisco.aesop.cli.commands.show_commands.class.

```
    <cmd>
        <pattern type=”id”>show</pattern>
        <help>Show running system information</help>
        <param>
            <pattern type="id">parser</pattern>
            <help>Display parser information</help>
        </param>
        <param>
            <pattern type="id">commands</pattern>
            <help>Display all the CLI commands</help>
        </param>

        <action paginate=”yes”>
            <class>com.cisco.aesop.cli.commands.show_commands.class</class>
        </action>
    </cmd>
```

pipehelp attribute (within type element)

```
<!ATTLIST action type (default) "default" paginate (yes|no) "no" pipehelp (yes|no) "no">
```

The `action` element supports the `pipehelp` attribute. The `pipehelp` attribute instructs the CLI server to add the pipe '|' token to the end of the command line. The pipe filters the text that is output from the eval( ) method in the handler class. (The handler class in the example below is com.cisco.aesop.cli.commands.show_commands.class.)

Pipe filters:

- begin—Begin with the line that matches.
- exclude—Exclude the lines that match.
- include—Include the lines that match.
- page—Paginate output (`--More--`).

pipehelp: Example

This example shows how to create a command which will implement the pipe feature.

The following example command is "`show parser commands`".

```
<cmd>
    <pattern type=”id”>show</pattern>
    <help>Show running system information</help>
    <param>
        <pattern type="id">parser</pattern>
        <help>Display parser information</help>
    </param>
    <param>
        <pattern type="id">commands</pattern>
        <help>Display all the CLI commands</help>
    </param>
    <action pipehelp=”yes”>
        <class>com.cisco.aesop.cli.commands.show_commands.class</class>
    </action>
</cmd>
```

Example:

```
hostname> show parser commands | ?
    begin Begin with the line that matches
    exclude Exclude lines that match
    include Include lines that match
    page Paginate output (--More--)
```

The output from "`show parser commands`" is piped through one of the filter types: begin, exclude, include, or page.

Example:

```
hostname> show parser commands | begin “d”
```

"`show parser commands`" invokes the eval( ) method in the handler class com.cisco.aesop.cli.commands.show_commands.class.

Each line of the output from the command is piped through "begin" filter type—in this example, only allowing commands starting with the letter "d".

class

```
<!ELEMENT class (#PCDATA)>
```

The `class` element, which is a child of the `action` element, specifies the handler class containing the eval( ) function/method invoked when the corresponding CLI command is recognized by the CLI parser. The name of the handler class contains the fully specified domain and class name (for example, com.cisco.aesop.service.handler.class). As a naming convention, we recommend that the mode of the CLI command is included in the class name. This prevents name-space collisions. For example, the "exec" mode is part of the name of the following class: com.cisco.aesop.service.exec_handler.class

class: Example

This example shows the `class` element used within the `action` element to define the eval( ) handler for the CLI command. The CLI command is "`show users`", and the `class` element is com.cisco.aesop.exec_show_users.class.

```
<cmd>
    <pattern type=”id”>show</pattern>
    <help>Show running system information</help>
    <param>
        <pattern type="id">users</pattern>
        <help>Display user information</help>
    </param>
    <action>
        <class>com.cisco.aesop.exec_show_users.class</class>
    </action>
</cmd>
```

interrupt-timeout

```
<!ELEMENT interrupt-timeout (#PCDATA)>
```

The `interrupt-timeout` element overrides the interrupt-timeout for this action. The parsed character data evaluates to an integer specifying the duration (in minutes) of the interrupt-timeout.

The following example is very similar to the previous example, except for the `interrupt-timeout` element that sets the interrupt-timeout to 10 minutes.

```
<cmd>
    <pattern type=”id”>show</pattern>
    <help>Show running system information</help>
    <param>
        <pattern type="id">users</pattern>
        <help>Display user information</help>
    </param>
    <action>
        <class>com.cisco.aesop.exec_show_users.class</class>
        <interrupt-timeout>10</interrupt-timeout>d>
    </action>
</cmd>
```

Type Regex

In the pattern tag, type= 'regex' defines the contents as being a regular expression conforming to the Perl 5 regex standard. See http://perldoc.perl.org/perlre.html for Perl 5 regex documentation.

Regex definition examples:

Example 1: A 1 or 2 digit number from 0 to 99.

```
<pattern type='regex'>[0-9][0-9]?</pattern>
```

Example 2: An alphanumeric string.

```
<pattern type='regex'>[a-zA-Z0-9]*</pattern>
```

Example 3: An HTTP URL.

```
<pattern type='regex'>http://[-*.+:@?=a-zA-Z0-9/_]*</pattern>
```

CLI Plug-in Invocation

To enter configuration mode and invoke CLI plug-in command, perform the following steps.

SUMMARY STEPS

1. **configure terminal**
2. **app-service** *application-name*

For configuration details, see "CLI Plug-in Invocation" in the *Cisco AXP User Guide*.

CLI Plug-in Action APIs

CLI plug-in action scripts or classes, supported by the distribution service, are written as C classes, Java classes, and Bash shell scripts. The API within the scripts/classes is provided by the eval( ) method (for Java), or the eval( ) function (for C).

After the user enters a CLI command, a CLI plug-in action corresponding to that particular command is invoked. The CLI plug-in action interacts with the application. For further details of the necessary CLI plug-in actions, application, and DTD, see the "CLI Plug-in Applications" section on page 56.

The CLI plug-in action API is implemented by using the eval( ) method or function. Compile the API calls into the appropriate C libraries or Java classes.

The APIs for a Java, C, or Shell script, that are required for the action class, are described below:

Java

The action class must implement the eval( ) method.

Compile Java action classes into files with a name format of **<action_name>.class**. The class can be in a namespace hierarchy; for example, com.xyz.myclass.class. The hierarchical class name must be defined in the XML definition for the CLI plug-in: see the class element of the "XML DTD Definition" section on page 61.

C

The action class must implement the eval( ) function.

Compile C action classes into a shared library with the name **<action_name>.so**.

Shell script

Shell scripts are directly executed using a C system call.

Since a shell script can only return an integer value indicating its exit status, there is no support for a shell script to pass back messages displayed on the CLI console.

The APIs use the following terms:

- grammar: space delimited string that specifies the syntax of the CLI command. For example, “show time”.
- tokens: space delimited string that specifies the actual CLI command typed by the user. For example, “show tim”, “sh time” or “sho ti”.
- return status: string that contains an integer specifying the CLI execution return code. 0 indicates success, non-zero indicates failure
- return display: string that is displayed on the CLI console to the user who types in the CLI command

APIs are explained as follows for Java, C, and Bash.

- eval: CLI plug-in Action API, page 70
- evalError: CLI plug-in Action API, page 73

Note The grammar[ ] parameter may contain a regular expression (regex). Cisco AXP uses the Perl5 regex. For further information, see the Perl documentation at: http://perldoc.perl.org/perlre.html.

eval: CLI plug-in Action API

Java

```
public static String[] eval(String tokens[], String grammar[])
```

Invokes a CLI command.

The example Java code MyTime.java, includes a call to the eval( ) method. It is similar to the Java code shown in the CLI plug-in of Figure 8 on page 58.

Parameters:

```
tokens[] - the space delimited string of tokens entered by the CLI user,
split into an array, e.g. "sho tim" is {"sho", "tim"}

grammar[] - the space delimited string of grammar matching the tokens,
split into an array. This grammar list is defined in the custom CLI
definition, e.g. "show time" is {"show", "time"}
```

Returns:

```
String[] - a String array containing 2 Strings. The first String is the
return status: "0" - succeed, non-zero - failed. For config CLIs, the
failed return status is not saved. The second String is the return
display. The rest of the list is ignored. For example, {"0", "Tue, 07 Aug
2007 16:20:36"}
```

Example:

```
eval({"sho", "tim"}, {"show", "time"});
  returns the following:
  {"0", "Tue, 07 Aug 2007 16:20:36"}
```

C

```
void eval(char* tokens[], int tokens_size, char* grammar[], int
grammar_size, char** ret_status, int* ret_status_size, char**
ret_display, int* ret_display_size);
```

Invoke a CLI command.

Parameters:

```
tokens[] – the space delimited string of tokens entered by the CLI user,
split into an array. For example, "sho tim" is represented as {"sho",
"tim"}

tokens_size – length of the tokens[] array. For example, 2 (for the above
tokens[] array.

grammar[] – the space delimited string of grammar matching the tokens,
split into an array. This grammar list is defined in the custom CLI
definition. For example, "show time" is represented as {"show", "time"}

grammar_size – length of the grammar[] array. For example, 2 (for the
above grammar[] array.

ret_status – string of return status: "0" – succeed, non-zero – failed.
For config CLIs, the failed return status is not saved.

ret_status_size – size of the ret_status string

ret_display – string of the CLI console display. For example, "Tue, 07
Aug 2007 16:20:36"

ret_display_size – size of the ret_display string
```

Returns:

```
N/A – the parameter ret_status and ret_display (see above) provides the
return values
```

Example:

```
my_time.c

#include ...

void eval(char* tokens[], int tokens_size, char* grammar[], int
grammar_size, char** ret_status, int* ret_status_size, char**
ret_display, int* ret_display_size){
...
}
...
eval({"sho", "tim"}, 2, {"show", "time"}, 2, "0", 1, "Tue, 07 Aug 2007
16:20:36", 25);
```

Returns: (in parameters ret_status and ret_display)

```
ret_status = "0"
ret_status_size = "1"
ret_display = "Tue, 07 Aug 2007"
ret_display_size = 25
```

Bash

```
my_time.sh $1 $2
```

The bash shell action is invoked differently than C and Java. Instead of invoking a function inside the shell script, the script itself is invoked, passing the tokens and grammar as the arguments. Therefore, this signature applies to invoking the shell script itself.

Parameters:

```
$1 - CLI token list enclosed with quotes ("), CLI params are delimited by
a space ( )

$2 - CLI grammar list  enclosed with quotes ("), CLI params are delimited
by a space ( )
```

Returns:

```
An integer value indicating whether the command succeeded.
  0 - succeed
  Non-zero - failed
For config CLIs, the failed return status is not saved. Since a shell
script can only return an integer value indicating its exit status, there
is no support for a shell script to pass back any message for the CLI
console to display.
```

Example:

Example script

```
my_time.sh

#!/bin/bash
# inside the shell script my_time.sh
...
tokens=$1
grammar=$2

# Do something with ${tokens} and ${grammar}
...

exit 0
```

Example script call

```
my_time.sh "sho tim" "show time"
returns the following:
0
```

evalError: CLI plug-in Action API

Java

```
public static void evalError()
```

Optional interrupt handler that handles interrupts during action invocation. Called in a separate thread spawned by the CLI Distribution Listener.

Parameters:

```
None
```

Returns:

```
None
```

Example:

```
public static void evalError() {
..
}
```

C

```
void eval_error()
```

Optional interrupt handler that handles interrupts during action invocation. Invoked by a signal raised from the CLI Distribution Listener main thread to the worker thread that invoked eval( ).

Parameters:

```
None
```

Returns:

```
None
```

Example:

```
void eval_error( ){
...
}
```

Supporting the *no* Prefix

Consider the following syntax:

```
>set timeout [0-9][0-9]?
```

The command may be prefixed with *no*, for example:

```
>no set timeout 10
```

The *no* prefix is automatically supported for all configuration CLIs and is appended in front of the CLI tokens in the tokens array.

In the following example, the CLI plug-in action receives the items shown in these Grammar and Tokens arrays:

Grammar:

```
set timeout [0-9][0-9]?
```

Tokens:

```
no set timeout 10
```

CLI Plug-in Distribution Listener APIs

An application starts the CLI plug-in distribution service by calling the CLI plug-in distribution listener to start within its main program.

Starting the distribution listener allows the CLI actions to be invoked within the same program space as the main program.

The CLI plug-in distribution listener can be started in a Java, C or shell script. Depending on the language being used, the code invokes one of the following APIs in its main program.

The following Java code snippet shows the call to the startCLIDistributionVMThread( ) method.

Java

```
package com.cisco.aesop.apphosting.cli_distribution.CLIDistributionVM;

public static void startCLIDistributionVMThread(String appName);
  Starts the CLI Plugin Distribution Service that is non-blocking.

Parameters:
  appName—Name of the application/virtual instance; this name must match
the name that is used for the packaging process.

Returns:
  n/a

Example:
  CLIDistributionVM.startCLIDistributionVMThread("myAppFoo");

make/compile-time dependency:
  cli_distribution_vm.jar

run-time dependency:
  cli_distribution_vm.jar
  localsocket.jar
  /cli_comm—path to the CLI actions
```

The following C code snippet shows the call to the start_cli_distribution_vm_thread( ) function.

C

```
cli_distribution_vm.h

int start_cli_distribution_vm_thread(char* app_name);
  Starts the CLI Plugin Distribution Service that is non-blocking.

Parameters:
  app_name—name of the application/virtual instance; this name must match
the name that is used for the packaging process.

Returns:
  rc—return code for starting the listener
  0—succeed
  1—failed

Example:
  int rc = start_cli_distribution_vm_thread("my_app_bar");

make/compile-time dependency:
  cli_distribution_vm.h
  libcli_distribution_vm.so

run-time dependency:
  libcli_distribution_vm.so
  liblocal_socket.so
```

The following shell script snippet shows the call to cli_distribution_vm.

Shell Script

```
/bin/cli_distribution_vm $1

Parameters:
  $1—Name of the application/virtual instance, this name must match the
name that is used for the packaging process.

Returns:
  n/a

Example:
  cli_distribution_vm my_app_bar &
```

CLI Plug-in Errors

Action classes communicate the return status of a command to the Cisco AXP server. The return status informs the server if the command succeeded or failed at execution. Action classes use a return string to communicate messages that must be displayed by the console to the user.

The return string is only supported in C and Java action classes.

The return status is supported by C action classes, Java action classes, and shell scripts.

Error messages are displayed on the CLI console when any of the following conditions are true:

- A CLI plug-in action returns a non-zero return status code.
- The Cisco AXP CLI server, CLI plug-in distribution service, or CLI plug-in actions fail to communicate with each other.
- A CLI plug-in action invocation times out.

A header in the error message indicates that the message is displayed by the CLI plug-in service.

Interrupting Action Invocation

The user of a CLI plug-in can interrupt action invocation by pressing Ctrl-C.

You have the option to implement an interrupt handler to handle the Ctrl-C interrupt for action classes. The interrupt handler is only available for C, and Java. See the "evalError: CLI plug-in Action API" section on page 73. The interrupt handler is action-specific—the handler only applies to a particular action class.

- In C, the interrupt handler is invoked as the same thread that has invoked the action. Raise a signal in the CLI Distribution Listener's main thread to the action invocation worker thread.
- In Java, the interrupt handler is invoked as a separate thread.

An action without an interrupt handler is dealt with by the CLI Distribution Listener. The listener uses a system call to stop the thread that is executing the eval( ) function.

The interrupt handler may become stuck. A rescue timer monitors the invocation of the interrupt handling, with a default value of 5 minutes. To change the timer value, change the interrupt-timeout in the XML definition file. The definition file is explained in the "XML DTD Definition" section on page 61. If the rescue timer detects a timeout, interrupt handling is regarded as having failed. If interrupt handling fails, the CLI Distribution Listener handles the interrupt.

Table 14 describes the different scenarios for interrupt handling.

Table 14 Interrupt Handling: Scenarios

Case	Action Contains Implementation of Interrupt Handler?	Interrupt Handler Returns from eval() Calls Immediately?	<interrupt-timeout> Element Defined as x Minutes?	Internal Interrupt Handling Description	Actions Experienced by the User
1	Yes	Yes	Yes	CLI invocation interrupted by eval_error.	CLI prompt returns, new CLI plug-in can be processed immediately.
2	Yes	Yes	No	CLI invocation interrupted by eval_error.	CLI prompt returns, new CLI plug-in can be processed immediately.
3	Yes	No	Yes	CLI invocation interrupted by distribution listener after x minutes.	CLI prompt returns, new CLI plug-in can not be processed until after x minutes.
4	Yes	No	No	CLI invocation interrupted by distribution listener after 5 minutes.	CLI prompt returns, and a new CLI plug-in can not be processed until after 5 minutes.
5	No	Yes	Yes	Case not possible.	Case not possible.
6	No	Yes	No	Case not possible.	Case not possible.
7	No	No	Yes	CLI invocation interrupted by CLI Distribution Listener immediately.	CLI prompt returns, and a new CLI plug-in can be processed immediately.
8	No	No	No	CLI invocation interrupted by CLI Distribution Listener immediately.	CLI prompt returns, and a new CLI plug-in can be processed immediately.

Simultaneous CLI plug-in Invocation

Multiple CLI sessions can be opened by a user. However, simultaneous CLI plug-in invocation to the same CLI Distribution Listener is not supported. When the CLI Distribution Listener is processing one CLI plug-in (that is, a CLI plug-in action is being invoked and has not yet returned), the listener blocks any new CLI plug-in request. Users see a message displayed on the CLI console saying that the CLI plug-in request is blocked, try again later.

Prepackaging the CLI Plug-in

The CLI plug-in application must be prepackaged before running the packaging tool. For more information on packaging, see the "Bundling Tool" section on page 97.

Prepackaging, using the CLI Plug-in SDK Packaging Tool, depends upon the setup of data in the following directory: `sdk/tools/custom_cli/`.

Each of the required components is listed below and then described in Table 15.

```
sdk/tools/custom_cli/
        actions/
        definitions/
        internal/
        cli_plugin.config
        cli_plugin.py
        cli_grammar.dtd
```

Note `cli_plugin.py` requires Java SDK version 1.4.2 or above to be installed on the development machine.

Table 15 Packaging Tool Components

Component	Description
`actions/`	Compiled CLI plug-in action classes.
`definitions/`	CLI plug-in definitions.
`internal/`	Internal files used by the tool. **Note** These files must not be modified.
`cli_plugin.config`	Configuration file.
`cli_plugin.py`	Executable file that packages the CLI plug-in application.
`cli_grammar.dtd`	Data Type Definition (DTD) file. **Note** This file must not be modified.

To prepackage the CLI plug-in application perform the following steps.

Step 1 Copy the action file into directory actions/ .

Step 2 Copy CLI plug-in definition files into directory definitions/.

Note The internal/ directory files must not be modified.

Step 3 Edit the configuration file cli_plugin.config. See Table 16.

Table 16 Configuration File Components

Component	Description
file	CLI plug-in definitions file, located in directory definitions/ .
javapath	Path to the directory Java SDK bin/ . J2SDK 1.4 is the recommended version of Java.
vm	Name of the application. This name must match the name provided later in the application packaging tool. See the "Package Name" section on page 80.
proj_src	Root directory from which this product will be prepackaged. • If proj_src argument is specified, your project file is moved to directory /source/showtimeapp/build, ready for packaging. • If proj_src argument is not specified, your project file is created in file showtime.tar in /source/showtimeapp/build. Later, during packaging, you will need to untar showtime.tar using for example: `tar xvf showtime.tar –C /source/showtimeapp/build`

Step 4 Run the CLI plug-in SDK packaging tool to package the definitions and actions.

To run the executable packaging tool, (cli-plugin.py), enter:

```
python cli-plugin.py
```

If the tool runs successfully, a tar file of the application is created in the output/ directory.

Step 5 Move the tar file to the source directory as shown in the following example.

Moving the tar File: Example

```
#cp output/showtime.tar /tmp/proj_dir/src
#cd /tmp/proj_dir/src
#tar xf showtime.tar
#rm showtime.tar
```

Each line of the example is explained below:

`#cp output/showtime.tar /tmp/proj_dir/src`
copy showtime.jar to source directory where the packaging tool runs

`#cd /tmp/proj_dir/src`
change directory to source directory /tmp/proj_dir/src

`#tar xf showtime.tar`
untar showtime.tar file to make it part of the source directory's contents

`#rm showtime.tar`
remove showtime.tar

Prepackaging of the CLI plug-in is now complete.

The next stage is to run the packaging tool. For more information, see the "Bundling Tool" section on page 97.

After packaging, to check the contents of a package use the script pkg_info.sh. For more information, see the "Package Information" section on page 43.

CLI Plug-in Application: Example

The development of a simple CLI plug-in application is shown in the following sections.

Health Status

We recommend that the health status is coded in an application in order for the **show state cli** command to show your application's health status.

If health status is not coded in an application, when the show state command is issued, the reported health status of an application is shown as dashes "---".

show state command (where the Application Health Status is not Coded): Example

```
my-javalin> app-service helloworld
my-javalin(exec-appservice)> show state
APPLICATION STATE HEALTH

helloworld online ---
```

Package Name

You must determine the package name for your application. The same package name must be used in the following three places:

1. Package name assigned to the VM topic in the cli_plugin.config file.
2. Package name assigned as an argument when the application calls the API **startCLIDistributionVMThread**( ).
3. Package name assigned to the **name argument of the packaging tool .

Show Time Application: Example

A custom command is defined by an XML definition file and an action script. You can create code for the custom command, package it with your application, and then import it into Cisco AXP.

As an example, consider an application which has to respond to a user request to display the time using the command **show time**. (The user request is shown as step 3 in Figure 9 on page 81.)

To develop the "show time" example, perform the following steps.

Step 1 Create a directory: **mkdir /source/showtimeapp**.

Step 2 Create an XML file **MyTime.xml** and save the file to directory **/source/showtimeapp**. See the "XML Script: Example" section on page 82.

Step 3 Create action class MyTime.java, see the "Action Class: Example" section on page 82.

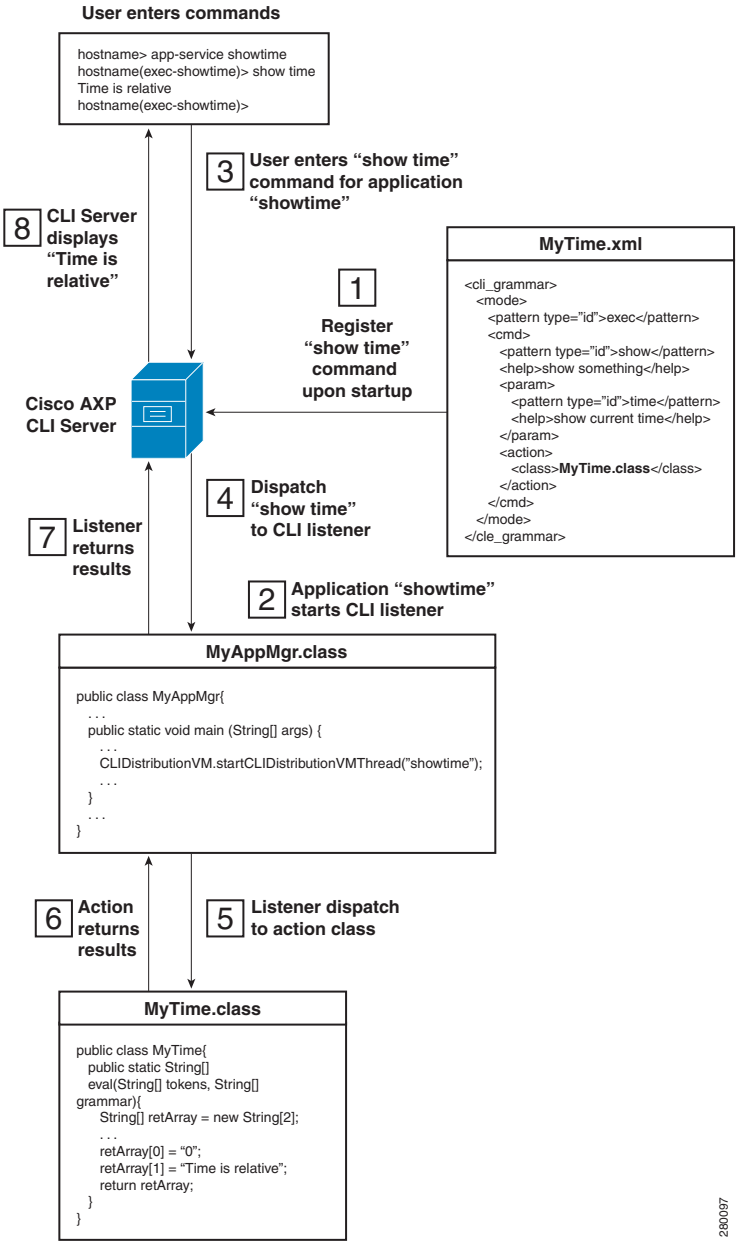
Step 4 Create an application (listener), see the "Starting a Listener from the Application: Example" section on page 83.

Step 5 Build and test, see the "Building and Testing: Example" section on page 84.

Step 6 Prepackage the CLI plug-in application, see the "Prepackaging the CLI Plug-in: Example" section on page 85.

Figure 9 describes the sequence of events when the showtime CLI plug-in application runs on the Cisco AXP CLI server.

Figure 9 CLI Plug-in Application: Example



XML Script: Example

Use the XML DTD to write an XML definition file. For further information on XML DTD definitions, see the "XML DTD File" section on page 59.

The following code script, MyTime.xml show time.

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE cli_grammar SYSTEM "cli_grammar.dtd" >

<cli_grammar>
  <mode>
    <pattern type="id">exec</pattern>
    <cmd>
      <pattern type="id">show</pattern>
        <help>show time</help>
      <param>
        <pattern type="id">time</pattern>
        <help>show current time</help>
      </param>
      <action>
        <class>MyTime.class</class>
      </action>
     </cmd>
  </mode>
</cli_grammar>
```

When the user enters command **show time**, Java action class **MyTime.class** is called.

Action Class: Example

MyTime.java returns the date and time. The eval() method is the API that accepts the user's CLI command and returns the current date and time. For a further explanation of the eval( ) method, see the "CLI Plug-in Action APIs" section on page 69.

```
/*MyTime.java is a simple CLI Action class that returns the application return code and
*the current date/time. If it fails an application return code of 1 is returned along
*with the exception message. The static main program can be invoked to easily test the
*program independently
*/
import java.util.Date;
import java.text.SimpleDateFormat;

/**
* Returns the current time of the day
*/

public class MyTime{
    public static String[] eval(String[] tokens, String[] grammar){
        String[] retArray = new String[2];
        try{
            if(tokens.length == 2 && grammar[0].tolower().equals("show") &&
            grammar[1].tolower().equals("time")){
                Date curTime = new Date();
                SimpleDateFormat sdf = new SimpleDateFormat("EEE, d MMM yyyy HH:mm:ss");
                retArray[0]="0";
                retArray[1]=sdf.format(curTime);
                }
            else{
                retArray[0]="1";
                retArray[1]="Unknown command";
            }
        }
```

```
        catch(Exception e){
            retArray[0]="1";
            retArray[1]=e.getMessage();
        }
        return retArray;
    }

    public static void main(String[] args){
        String[] tokens = {"show","time"};
        String[] grammer = {"show","time"};
        String[] message = eval(tokens,grammer);
        System.out.println("Exit code: " + message[0] + "  Result:" + message[1]);
    }
}
```

The XML script MyTime.xml defines the **show time** command. See the "XML Script: Example" section on page 82.

After compiling and testing the above action class (MyTime.class), store it in file showtime.jar so that it can be easily accessed later. Place the jar file in the /source/showtimeapp directory:

```
jar -cvf showtime.jar MyTime.class
```

When the user enters command **show time**, this action class MyTime.class processes the command. See the "Action Class: Example" section on page 82.

For the class MyTime.class to be called, a listener must be started in your application. See the "Starting a Listener from the Application: Example" section on page 83.

Starting a Listener from the Application: Example

Start an application listener using the thread startCLIDistributionVMThread. For example, the following command is used within the application MyAppMgr.

```
CLIDistributionVM.startCLIDistributionVMThread("showtime");
```

MyAppMgr gives the health status of the application, and activates a VM thread which listens for the defined **show time** command. When the user enters the **show time** command, the application calls the MyTime class.

```
/*
* Third party application that reports its health status, starts up the CLI distribution
* thread, and then waits.
*/
import com.cisco.aesop.apphosting.cli_distribution.CLIDistributionVM;
import java.io.*;
import java.lang.*;

public class MyAppMgr{
/**
*Displays and writes passed in message
*@param msg: message to be displayed and logged.
*@param pw: PrintWriter object
*@return nothing
*
*/
    public void printMessage(String msg, PrintWriter pw){
        System.out.println(msg);
        if(pw != null){
            pw.println(msg);
            pw.flush();
        }
    }
```

```
    /**
    *Sends the health status.  Logs exception.
    *@param name: application package name
    *@param status: health status
    *@param pw: PrintWriter object
    *@return nothing
    *
    */

    public void sendStatus(String name, String status, PrintWriter pw){
        try{
            Runtime rt = Runtime.getRuntime();
            rt.exec("/bin/app_status_notifier " + name + " " + status);
        }
        catch(Exception e){
            printMessage("Send of status failed.  Exception " + e.getMessage(),pw);
        }
    }

//code for testing application
    public static void main(String[] args){
    MyAppMgr mgr = new MyAppMgr();
    PrintWriter pw = null;
    try{
        pw = new PrintWriter(new FileOutputStream(new File("/var/log/showtime.log")));
        mgr.sendStatus("showtime","INITIALIZING",pw);
        mgr.printMessage("Starting VM Thread",pw);
// The following line starts the listener
        CLIDistributionVM.startCLIDistributionVMThread("showtime");
        Object obj = new Object();
        synchronized(obj){
            mgr.sendStatus("showtime","ALIVE",pw);
            mgr.printMessage("Going into wait mode",pw);
            pw.flush();
            obj.wait();
        }
    }
    catch(Exception e){
        mgr.printMessage("Exception " + e.getMessage(),pw);
    }
    finally{
        mgr.printMessage("Exiting MyAppMgr program",pw);
        mgr.sendStatus("showtime","DOWN",pw);
        if(pw != null){
            pw.close();
        }
    }
    } //end main
} //end class
```

Building and Testing: Example

Build

```
[bash]# javac -classpath cli_distribution_vm.jar MyAppMgr.java
```

Add this class to file showtime.jar, which allows for easy access later.

```
jar -uvf showtime.jar MyAppMgr.class
```

Test

For testing, you need access to the following Linux library: liblocal_socket.so.

Copy showtime.jar to directory /source/showtimeapp.

The java command below tests application MyAppMgr. File liblocal_socket.so and jar files are in directory “lib” —one level below the current directory.

Note You must be user “root” to perform this test in your development environment.

Note Provided your application (MyTime) is packaged with a dependency on the CLI plug-in package, then showtime.jar is the only one to be packaged. This is because the file liblocal_socket.so and other jar files are in the guest OS (virtual instance).

```
[bash]# java -Djava.library.path=lib/ -cp
lib/cli_distribution_vm.jar:lib/localsocket.jar:lib/showtime.jar MyAppMgr
```

Output:

```
Starting VM Thread
Going into wait mode
CLI_PLUGIN:CLIDistributionVM.receiveIPCMessages: server socket opened successfully
```

Create a build directory

Create a directory to hold prepackaged files to be packaged and installed onto Cisco AXP:

```
mkdir /source/showtimeapp/build
```

Prepackaging the CLI Plug-in: Example

The following example of prepackaging a CLI plug-in application follows the steps in the “Prepackaging the CLI Plug-in” section on page 78.

Step 1 Copy the action file MyTime.class into directory actions/.

Step 2 Copy CLI plug-in definition file MyTime.xml into directory definitions/.

Step 3 Edit the configuration file cli_plugin.config.

```
file=MyTime.xml
javapath=/usr/bin/java/bin
vm=showtime
proj_src=/source/showtimeapp/build
```

You have the option of whether to change the proj_src argument or not. See proj_src in Table 16 on page 79.

Step 4 Run the CLI plug-in SDK packaging tool to package the definitions and actions.

```
python cli-plugin.py
```

For further information, see step 4 of the “Prepackaging the CLI Plug-in” section on page 78.

Next, move the tar file to the source directory, see the “Moving the tar File: Example” section on page 79.

Prepackaging is complete.

Troubleshooting and Logging

This section explains some troubleshooting tips and the main types of logging in the following sub-sections:

Note To return to the Table of Contents, click here.

Troubleshooting Your Application

Debugging Steps

The following debugging steps go through a range of options for debugging your application.

Step 1 To check that your application is running:

app-service <app name>

where <app name> is the name of your application

show process

Shows all processes sorted by process ID in ascending order. See "Viewing Processes" in the *Cisco AXP User Guide*. If your application is part of a CLI plug-in application, the application must always be running.

Step 2 To show the state of your application, enter the following:

show state

Step 3 If application state is not "online" go to Step 4.

If application health = "ALIVE", then your application is running, go to the end of the steps.

If application state is "---" and your application is coded to provide its health status, go to step 5.

If application health NOT = "ALIVE", then go to step 5.

Step 4 To find out more about why the virtual instance for your application does not start, perform the following steps.

a. Exit to the Cisco AXP prompt by entering **exit**

b. Check the messages.log file or messages.log.prev file: enter one of the following commands:

- **show log name messages.log containing <application name> paged**
- **show log name messages.log.prev containing <application name> paged**

, where **<application name>** is the name of your application. The log files contain error messages created by the system.

Look at the log files displayed by one of these two commands. There should be evidence of why the virtual instance did not start. Log files are displayed one page at a time.

Step 5 Provided that you have configured your application for logging, the application log files are located in your guest instance and contain messages written by your application. Enter one of the following commands:

- **show log name messages.log paged**
- **show log name messages.log.prev paged**

, where **<application name>** is the name of your application. Look at the log files displayed by one of these two commands. There should be evidence of why your application did not start. The log files contain error messages created by your application.

Note If the application is writing log files to the directory /var/log or has a link which points to this directory you can view these log files by specifying their name in instead of messages.log in the first of the two commands above.

Step 6 Turn on tracing for your application. You can trace specified modules, entities, or activities. For full tracing, enter:

trace all

After your process has run, enter:

show trace buffer

Step 7 For intermittent issues, Cisco AXP provides support for Linux commands such as **vmstat** and **top** in the shell so you can check if your application is CPU intensive or using up a lot of memory. Using up available RAM or allocated CPU may cause the application to function very slowly. To check that your system is not exceeding the traffic capacity, enter the following command on the router:

show interfaces <interface>

vi Editor Troubleshooting

The vi editor does not behave as expected with some terminals. To fix this, run vi with parameter:

-T ansi

Logging

For logging inside a virtual instance, the syslog daemon *syslog-ng*, adds tracing or logging information to file /var/log/messages.log.

Logging is described in the following sections:

- Log File Rotation, page 88
- Logging Levels, page 88
- Logging Usage, page 89

Note To return to the Table of Contents, click here.

For information about the following topics, see the *Cisco AXP User Guide.*

- Configuring Log File Size Limits
- Configuring System Log Levels
- Configuring Remote Logging.
- Viewing Files
- Copying Files
- Clearing Files

Log File Rotation

The Cisco AXP syslog supports log file rotation of two files:

- messages.log (primary log file)
- messages.log.prev (backup log file)

When the messages.log reaches a limit specified by the **limit log-file size** command, the file's contents are moved to backup file messages.log.prev and a new messages.log file is started. The default log file size is 20 MB.

You can use the command **show logs** to display the contents of all log files, and the command **show log name** to display the contents of a log file in the /var/log directory.

Logging Levels

Cisco AXP supports the following log levels:

- LOG_EMERG: system is unusable
- LOG_ALERT: action must be taken immediately
- LOG_CRIT: critical condition
- LOG_ERR: error condition
- LOG_WARNING: warning condition
- LOG_NOTICE: normal, but significant, condition
- LOG_INFO: informational message
- LOG_DEBUG: debug-level message

Out of all the log levels that are available to the application, the CLI reduces these to a more manageable number of four log levels.

You can set your log threshold to one of the following values:

- **Errors**: Events with LOG_ERR and lower severity are logged, including LOG_EMERG, LOG_ALERT, and LOG_CRIT.
- **Warning**: Events with LOG_WARNING and lower severity are logged, including all error messages described in the error threshold.
- **Notice**: Events with LOG_NOTICE and lower severity are logged, including all messages described in the warning threshold.
- **Info**: Events with LOG_INFO and lower severity are logged, including all messages described in the notice threshold.

Logging Usage

Logging on the virtual instance is explained below for either C or Java code:

- C: Syslog, page 89
- Java: Log4j, page 89

Log messages go to file /var/log/messages.log in the virtual instance. After the log file reaches its specified limit (set by command `limit log-file size`), the contents of the log file are moved to a backup log file messages.log.prev. A new messages.log file starts to collect log messages. The default log file size for the log file and the backup log file is 5MB.

C: Syslog

For more information and examples on syslog APIs for C, see the GNU website.

C Syslog: Example

```
int main(int argc, char* argv[]){
    setlogmask (LOG_UPTO (LOG_DEBUG));
    openlog("myapp", LOG_CONS | LOG_PID | LOG_NDELAY, LOG_LOCAL1);
    syslog(LOG_DEBUG, "myapp_user_management.eval() processing a CLI message\n ");
}
```

Java: Log4j

For more information on the Log4j package regarding syslog, refer to **syslog appender** in the log4j documentation at: http://logging.apache.org/log4j/docs/documentation.html.

In Cisco AXP APIs and libraries, logs are redirected to stdout (via System.out.println).

To redirect logs to syslog, pipe the application's output to /bin/logger, as shown in the following example.

Redirect logs to syslog: Example

This example starts application MyAppMain, and uses a CLI plug-in. The stdout of the CLI plug-in and the output from System.out.println calls made by the application are redirected to syslog with a log level of LOG_INFO.

```
# java -cp
./app_bin/myApp.jar:/cli_comm/:/usr/lib/java/localsocket.jar:/usr/lib/java/cli_distributio
n_vm.jar com.myApp.MyAppMain | /bin/logger -p info
```

Turn on logging: Example

```
Logger myLogger = Logger.getInstance("myapp");
myLogger.debug("myapp_user_management.eval() processing CLI message");
```

Enable syslog appender: Example

```
# To enable the syslog appender, set the root logger level to DEBUG and an appender to
mySyslog.
log4j.rootLogger=DEBUG, mySyslog, myFile, myConsole

# appender definitions
log4j.appender.myConsole=org.apache.log4j.ConsoleAppender
log4j.appender.mySyslog=org.apache.log4j.net.SyslogAppender
log4j.appender.myFile = org.apache.log4j.RollingFileAppender
```

```
# myConsole config
log4j.appender.myConsole.layout=org.apache.log4j.PatternLayout
log4j.appender.myConsole.layout.ConversionPattern=%p %t %c - %m%n

# mySyslog config
log4j.appender.mySyslog.layout=org.apache.log4j.PatternLayout
log4j.appender.mySyslog.layout.ConversionPattern=%p %t %c - %m%n
log4j.appender.mySyslog.SyslogHost=localhost
log4j.appender.mySyslog.Facility=USER
log4j.appender.mySyslog.threshold=INFO

# myFile config
log4j.appender.myFile.File = example.log
log4j.appender.myFile.MaxFileSize = 100KB
log4j.appender.myFile.MaxBackupIndex=1
log4j.appender.myFile.layout=org.apache.log4j.PatternLayout
log4j.appender.myFile.layout.ConversionPattern=%p %t %c - %m%
```

Syslog Server Logging

To log messages to the host syslog server, use the syslog server feature. For example, messages can be logged from a workstation to the host syslog server.

Logging messages to the host syslog server for either C or Java code, is explained in the following sections.

- C: Syslog Server, page 90
- Java: Syslog Server, page 90

C: Syslog Server

In C programs, to direct log messages to the host syslog server, configure the syslog server config file, located at either /etc/syslog.conf, or /etc/syslog-ng.conf. These location names apply if the virtual instance's syslog server is syslog-ng. See the "Logging" section on page 87.

Redirect: Example

To redirect all log messages with facility= "local1", and priority value of "info" or higher, append the following line to the syslog server config file:

```
local1.info @myblade
```

Java: Syslog Server

In Java programs, to direct log messages to the host syslog server, edit the log4j.properties file in the log4j package (see the "Java: Log4j" section on page 89). For example,

```
log4j.rootLogger=DEBUG, mySyslog

# mySyslog config
log4j.appender.mySyslog.layout=org.apache.log4j.PatternLayout
log4j.appender.mySyslog.layout.ConversionPattern=%p %t %c - %m%n
log4j.appender.mySyslog.SyslogHost=myblade
/* Note the SyslogHost is "myblade" compared to "localhost" used by logging in the virtual
instance */
log4j.appender.mySyslog.Facility=USER
```

```
log4j.appender.mySyslog.threshold=INFO
```

In addition to changing your application code, commands must be issued on the Cisco AXP service module to run the host syslog server:

```
blade# config terminal
(config)# syslog-server
```

To show the remote syslog server status:

```
blade# show syslog-server
Syslog Server
Status:                 RUNNING
```

To configure the number of files for rotation and the file size:

```
blade# config terminal
(config)# syslog-server limit file-rotation 2 file-size 500
```

`file-rotation` ranges in value from 1 to 40, with a default of 10.

`file-size` ranges in value from 1 to 1000 MB, with a default of 20 MB.

Bundling and Packaging

Bundling is the activity of grouping together packages, ready for installation on the Cisco AXP service module. This section contains information about the following:

- Packaging, page 91
- Bundling, page 96

Note To return to the Table of Contents, click here.

Packaging

Packaging Tool

Note Before running the packaging tool, check that your PATH environment variable has the system paths listed at the beginning of the string. To determine the value of your path environment variable, enter:

```
>echo $PATH
```

For example, a path similar to the following path value is displayed:

```
>/bin:/sbin:/usr/bin:/usr/local/sbin:/usr/sbin
```

Call the packaging tool command (pkg_build.sh) using one of the following two modes:

1. **Single Command Line Mode:** Enter the command **pkg_build.sh <list of arguments>**.
2. **Interactive Mode:** Enter command **pkg_build.sh** and then enter each argument separately (one argument on each line).

Note If you are packaging bash scripts, the packaging tool expects the bash scripts to be version 3.0 or above.

The values supplied to the packaging tool determine the resources allocated to the packaged application when the package is installed. The only way to change the resource allocation is by re-packaging.

Upgrading

When you are upgrading an application, there are three arguments/parameters of the packaging tool to consider:

1. uuid: enter the uuid of the package when it was first created. To recall the uuid that was used, use the command "show software versions detail". (See "Verifying and Troubleshooting System Status" in the *Cisco AXP User Guide*.)
2. name: enter the name of the package using the same name that was used when the package was originally created.
3. version: the upgraded application must have a different version compared to the application on the service module.

Other arguments include: deps, disk-limit, and cpu-limit. Details of these arguments are found in the following sections.

Note After creating a package, you can check the contents of a package by using pkg_info.sh, see the "Package Information" section on page 43.

The Packaging tool is described in the following sections:

- Packaging Tool Arguments
- Dependencies, page 94
- Disk Limit, page 95
- CPU Limit, page 95
- Packaging Tool Command: Example, page 95

Packaging Tool Arguments

The arguments to the packaging tool are described in Table 17.

Table 17 Packaging Tool Arguments

Argument	Brief Description	Description	Example
project-dir	Project directory	Contains package files, templates, and temporary files created during packaging.	
dev-cert	Location of development certificate	X.509 certificate authorized for software development on Cisco AXP.	/xyz_source/x509.cer
dev-auth	Location of software development authorization file	Binary file that grants permission to the certificate holder to develop software for Cisco AXP.	/xyz_source/dev_auth

Table 17 Packaging Tool Arguments (continued)

Argument	Brief Description	Description	Example
private-key	Location of private key file	File containing the RSA key that is used to sign the application third party software.	/xyz_source/privkey.txt
name	Name of application	Name of the packaged application.	myapp
version	Version of application	Version of the packaged application.	1.0
description	(Optional) Description of application	Description of the application's function.	myapp utility
uuid	(Optional) Unique identifier of the application	1. You can enter a uuid value; for example a uuid generated using the Linux "uuidgen" tool. This is recommended, see the Note below this table. or 2. You can leave the uuid as blank: an ID is automatically generated.	
source-dir	Location of application files	Directory that contains application files to be packaged. The directory structure should mirror the desired directory structure on the target file system.	/xyz_source/root_fs
protect	(Optional) List of files to be protected	Plain text file containing a list of files (one file per line) that will be checked for tampering before application initialization.	
deps	(Optional) Package Dependencies	See Dependencies, page 94.	<dependency list>; for example see Dependencies, page 94
disk-limit	Disk Limit	Minimum disk space required by the application. For further information on disk-limit, see Disk Limit, page 95.	10M
memory-limit	Memory Limit	RAM usage limit. Consider the target hardware resources when selecting a memory-limit. The application may fail to install or run if the requested amount of resources are not available on the target hardware. (1 M = 1048576 bytes.)	10M

Table 17 Packaging Tool Arguments (continued)

Argument	Brief Description	Description	Example
cpu-limit	CPU Limit	CPU resources utilized by the application. For a description of the CPU limit, see CPU Limit, page 95.	200
postinstall	(Optional) Postinstall script	(Optional) Script to run after installation. Specify the location of the script using a relative path with respect to the common root directory. For example, if script postinstall.sh is in directory /source/build/bin, and the common root directory is /source/build, specify the location of the script as /bin/postinstall.sh	/bin/postinstall.sh

Dependencies

The list of dependencies parameter (deps) consists of a selection of available packages from the SDK and project directories. When listing a package that already exists on the development workstations, you can specify the UUID and filename of that package. After each dependency, the user must specify one of the following dependency types:

- From [version]: Dependency must be of the specified or later version.
- To [version]: Dependency must be of the specified or earlier version.
- Exclude [version]: Package cannot be installed with other dependencies for the specified version.
- Only [version]: Dependency must be of the specified version.
- None: Package cannot be installed with the specified dependency.
- All: Dependency of any version must be present for the package installation.

Version

Each application package has a version number, which is used by the installer when resolving dependencies between subsystems.

For dependency checking only the first two digits of the version number are checked.

The first digit is treated as Major version number and the second one is treated as Minor version number.

For example, version 1.2.3.4 has a major number of 1 and minor number of 2, and is treated the same as version 1.2, or version 1.2.3 for dependency checking.

List of Dependencies: Syntax

[Dependency List] => (One or more colons (":") separated [Dependency String] [Dependency String] => [Dependency UUID],(One or more comma (",") separated [Dependency Type] [Dependency Type] => [to/from/exclude/only/none/all](=[version])

List of Dependencies: Example

```
--deps 'cc1b5b06-6b17-42c9-b9dd-88c29674b390,from=1.0,to=2.0:
b569b32d-1b5b-4306-8eb6-c7d6808aa7b6,only=3.0'
```

Disk Limit

When calculating disk-limit, consider the disk space required by the following components: application package (not including the disk space used by packages from Cisco), log files, and core dumps.

The disk limit is the minimum disk space required for the application. When disk resources falls below this limit, you will not be able to install the application. The disk limit must be specified in megabytes, for example 1M.

CPU Limit

The cpu-limit argument for the packaging tool sets the CPU utilization of the application package measured in CPU index points.

The cpu-limit is the minimum CPU limit for the application. The minimum CPU limit specifies the minimum amount of CPU resources allocated for the application. When the available CPU resources on the target service module fall below the CPU limit, you will not be able to install the application.

The CPU index for the service module is a known value, calculated using a benchmark. A more powerful service module will have a higher CPU index and will be able to accommodate a higher number of concurrent applications. For further information, see the "Dedicated Application Resources" section on page 8.

A CPU index is assigned to each service module based on the power of the CPU. The application's CPU index is then calculated, based on the following formula:

Application CPU index = CPU Index(of service module) * CPU usage%

For example, the NME APPRE 302 service module has a CPU index of 10000. Each service module has a CPU index relative to a value of 10000. If an application requires 60% CPU utilization, the applications's CPU index is 10000 * 60% = 6000. Supply this value as the cpu-limit argument.

The advantage of knowing the application CPU index is that the application package can be installed on other service modules, and you know it will run successfully if the module has enough CPU resources.

Packaging Tool Command: Example

```
echo $PATH
//the next line shows example output
/bin:/sbin:/usr/bin:/usr/local/sbin:/usr/sbin

/source/axp_tools/appre-sdk.nme.1.0.1/tools/pkg_build.sh --project-dir
'/source/helloworld_app/package' --dev-cert '/source/helloworld_app/certs/dev_certificate'
--dev-auth '/source/helloworld_app/certs/dev_authorization' --private-key
'/source/helloworld_app/certs/private.key' --name 'helloworld' --version '1.0'
--description 'hello hello' --source-dir '/source/helloworld_app/build' --disk-limit '10M'
--memory-limit '10M' --cpu-limit '100' --postinstall '/bin/post-install.sh'
```

Bundling

Bundling refers to grouping together packages, using the bundling tool. The bundles may include the Cisco AXP Host Operating System. Information on bundling is in the following sections:

Note If the signed infrastructure add-on packages (developed by Cisco) are not bundled with signed add-on applications (developed by third party), then you will not be able to install the bundle on the service module. That is, you cannot install a bundle consisting only of packages developed by Cisco.

Bundling with the Cisco AXP Host OS

To install bundles including the installation of a third party signed add-on application on the service module, including the Cisco AXP host OS, use the command: **software install clean**.

After a clean installation, the startup configuration is cleared back to the factory default. (This is similar to the effect of a **write erase** command.)

The following list shows examples of bundles that include the Cisco AXP host OS.

- Cisco AXP host OS.
- Cisco AXP host OS + third party signed add-on packages.
- Cisco AXP host OS + Cisco signed infrastructure add-on packages + third party signed add-on packages. (The third party signed add-on packages may have dependencies on Cisco signed infrastructure add-on packages.)
- Cisco AXP host OS + Cisco signed infrastructure add-on packages + third party signed add-on packages. (The third party signed add-on applications may have dependencies on Cisco signed infrastructure add-on packages and/or third party signed add-on packages).

Permitted Dependency

A third party signed add-on package may depend on a Cisco signed add-on package and a third party application package.

Non-permitted Dependencies

A third party signed add-on package (for example, signed by one third party—A) cannot depend on a third party signed add-on package (for example, signed by a different third party—B).

A third party signed add-on package cannot depend on a third-party add-on package—a chain dependency.

A third party signed application package cannot depend on a third party signed add-on package.

A third party signed add-on package cannot depend on more than one application package at a time.

Note Software installation authorization can only be applied to an X.509 certificate already signed by a Certificate Authority and is not part of the certificate.

Note The third party is responsible for providing their X.509 certificate to Cisco and for including Cisco's authorization checksum in the application package.

Bundling without the Cisco AXP Host OS

To install a bundle onto the application service module where the bundle does not contain the Cisco AXP host OS, and contains a third party signed add-on package, use the command:

software install add

The following list shows examples of bundles that do not include the Cisco AXP host OS.

- Third party signed add-on application packages.
- Cisco signed infrastructure add-on packages + third party signed add-on application package with or without dependencies on signed infrastructure add-on packages (developed by Cisco).
- Signed infrastructure add-on packages (developed by Cisco) +
  third party signed add-on applications (The third party signed add-on applications may have dependencies on Cisco signed infrastructure add-on packages and/or third party signed add-on packages) +
  third party signed add-on packages.(The third party signed add-on packages may have dependencies on Cisco signed add-on packages and/or signed third party add-on application packages.)

A third party add-on package may be bundled with its respective third party base application(s).

For dependencies limitations for third party add-on packages, see the "Non-permitted Dependencies" section on page 96.

Bundling Tool

The bundling tool (pkg_bundle.sh) compiles multiple packages into a single bundle.

Use the bundling tool by entering a single command line with a list of arguments. See Table 18.

Table 18 Bundling Tool Arguments

Parameter	Brief Description	Description
help	Version and usage	
project-dir	Project directory	Stores generated package files, templates, and temporary files created during packaging in a sub-directory given by adding the project-dir parameter and "/pkg". See Bundling Tool: Example, page 98.

Table 18 Bundling Tool Arguments

Parameter	Brief Description	Description
private-key	Location of private key file	File containing the RSA key used for security headers and package specification.
output	Output package	List of packages. The first name contains the name of the combined package "bundle". If the path name is not included or is invalid, the package is written to directory: project-dir/pkg. Second and subsequent names in the list of parameter values following "output" are the names of other packages forming the bundle. Payload files such as prt1, prt2 should be located in the same directory as their corresponding package files.

Bundling Tool: Example

In the following example, the two packages iosapi_test.1.1.pkg and axp-iosapi.nme.1.0.4.pkg are bundled together into package iosapi_test_bundle.1.0.4.pkg. The bundle is placed in project directory /iosapi_app/package/pkg.

Note If you specify the directory /iosapi_app/package in the --project-dir parameter, the package is placed in /iosapi_app/package/pkg.

```
/axp-sdk.1.0.4/tools/pkg_bundle.sh --project-dir '/iosapi_app/package' --private-key
'/certs/private.key' --output 'iosapi_test_bundle.1.0.4.pkg'
'/iosapi_app/package/pkg/iosapi_test.1.1.pkg' '/bryce/axp-iosapi.nme.1.0.4.pkg'
```

Packet Analysis

Traffic Filtering, and Packet Analysis with Cisco AXP are explained in the following sections:

- RITE Traffic Filtering and Sampling, page 99
- Packet Monitoring and Analysis, page 101

Note To return to the Table of Contents, click here.

RITE Traffic Filtering and Sampling

RITE (Router IP Traffic Export) is supported by Cisco AXP. RITE is a Cisco IOS software export tool that supports traffic filtering and sampling. RITE duplicates and exports traffic on a monitored interface. The Integrated-Service-Engine interface used by the Cisco AXP service module behaves like an Ethernet interface and is supported by RITE. Using RITE, the analysis module does not need to be attached to the Cisco ISR (unlike with the Cisco AXP Promiscuous Packet API).

RITE provides:

- Inbound, bidirectional, or outbound traffic monitoring
- Specification of the export interface
- Control of ACL filtering and sampling rate
- Ability to monitor interfaces with different encapsulation types including the Integrated-Service-Engine interface—behaves like an Ethernet interface
- SpecifIcation of vlan as the interface
- Sampling of one in every *n* packets during traffic sampling
- Easy profile capture before applying profile on an interface
- More granular traffic capture compared to using Cisco AXP Promiscuous Packet API
- You can specify ACL, sampling rate, and also the direction (inbound/outbound) of the captured traffic

There is however one limitation when using RITE: router generated traffic will not be captured by RITE. Router generated packets are process switched and not cef switched.

Configuring RITE

For additional information on configuring RITE, see “Configuring Router IP Traffic Export” in the *Cisco AXP User Guide*.

Note Router generated traffic is not captured by RITE because router generated packets are process switched and not cef switched. Also, there can be only one export interface per profile and one profile per monitored interface.

Configuring RITE is explained in the following three sections:

- Creating an RITE Capture Profile, page 99
- Configuring the RITE Capture Profile Parameters, page 100
- Applying the RITE Capture Profile to an Interface, page 101

Creating an RITE Capture Profile

Create a profile:

```
ip traffic-export profile <your-profile-name>
```

Configure the interface for export:

```
interface Integrated-Service-Engine 1/0
```

Configure the mac-address of the Integrated-Service-Engine1/0 interface:

```
mac-address <address>
```

Note In the above command, **mac-address** is the valid mac-address of the Cisco AXP service module.

However, since the service-module is configured using Cisco AXP Promiscuous Packet API, if the command is issued with an invalid mac-address, the command also works.

MAC Address a.a.a

The Cisco AXP service module is set to route packets by default, configuring a valid mac-address causes the routing stack of the service-module to route traffic. This leads to the situation where duplicated packets are routed. To avoid the routing of duplicated packets, we recommend that you configure an invalid mac-address. An example of an invalid mac-address is a.a.a. Using an invalid mac-address means that packets reach the Cisco AXP service module but are not routed.

If you want to capture both inbound and outbound traffic, configure the interface as bidirectional. (The default is inbound only.)

Creating an RITE Capture Profile: Example

```
jc1(config)#ip traffic-export profile rite-bi
jc1(conf-rite)#?
IP traffic export profile configuration commands
  bidirectional  Enable bidirectional traffic export
  exit           Exit from ip traffic export profile sub mode
  incoming       Configure incoming IP traffic export
  interface      Specify outgoing interface for exporting traffic
  mac-address    Specify ethernet address of destination host
  no             Negate or set default values of a command
  outgoing       Configure outgoing IP traffic export

ip traffic-export profile rite-bi
interface Integrated-Service-Engine1/0
    bidirectional
    mac-address a.a.a
!
```

Configuring the RITE Capture Profile Parameters

Configure the RITE capture profile by configuring the export interface, outgoing mac-address, and access list or sampling parameters.

Configure the incoming or outgoing traffic sampling as being access-list or sample rate.

- access-list: regular ACL
- sample: sample rate of the export traffic

Configuring the RITE Capture Profile Parameters: Example

```
jc1(config)#ip traffic-export profile rite-bi
jc1(conf-rite)#incoming ?

access-list  Apply standard or extended access lists to exported traffic
sample       Enable sampling of exported traffic


ip traffic-export profile rite-bi
  interface Integrated-Service-Engine1/0
```

```
  bidirectional
  incoming access-list 4
  outgoing access-list 4
  mac-address a.a.a
  incoming sample one-in-every 2
  outgoing sample one-in-every 2
!
```

Applying the RITE Capture Profile to an Interface

To monitor an interface, you must apply the profile on the interface:

```
ip traffic-export apply <profile name>
```

Applying the RITE Capture Profile to an Interface: Example

```
interface GigabitEthernet0/0
 description $ETH-LAN$$ETH-SW-LAUNCH$$INTF-INFO-GE 0/0$
 ip address 1.100.50.221 255.255.255.0
 ip traffic-export apply rite-bi
 duplex auto
 speed auto
 no keepalive
end
```

Packet Monitoring and Analysis

Packet monitoring for Cisco AXP is similar to packet monitoring in a Cisco IOS environment.

To enable packet monitoring on a Cisco AXP service module interface, see “Packet Analysis” in the *Cisco AXP User Guide*. (e.g This uses the **analysis-module monitoring** command in interface configuration mode.)

The Cisco AXP Promiscuous Packet API exports inbound and outbound traffic on the configured interface and exports local router-generated packets. See the “Promiscuous Packet API” section on page 101.

Promiscuous Packet API

The kernel is configured with the following directly accessible features:

- Raw socket interface (CONFIG_RAW).
- Raw socket memory mapped mode (CONFIG_RAW_MMAP).
- Socket filtering (CONFIG_FILTER).

The raw socket interface is compatible with the higher level abstraction of libpcap and MMAP packet access. For more information on libpcap/tcpdump, refer to:

- libpcap/tcpdump: http://www.tcpdump.org
- manual pages; for example, **man 3 pcap**
- tutorial:

  http://yuba.stanford.edu/~casado/pcap/section1.html

Note An application running inside a virtual instance can use a raw socket to monitor all the available network interfaces on the Cisco AXP service module.

Raw Socket: Example

```
#include <stdio.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <linux/if_ether.h>
#include <linux/if_arp.h>
#include <errno.h>

int main (int argc, char* argv[]){
  int result = 0;
  int s = socket(AF_PACKET, SOCK_RAW, htons(ETH_P_ALL));
  if(s != -1){
      struct sockaddr_ll socket_address;
      socket_address.sll_family = AF_PACKET;
      socket_address.sll_protocol = htons(ETH_P_ALL);
      bind(s, (struct sockaddr*) &socket_address, sizeof(socket_address));
      char* buffer= (char*)malloc(ETH_FRAME_LEN+1);
      int length=recvfrom(s, buffer, ETH_FRAME_LEN,0,NULL,NULL);
   }
   else{
     printf("Socket creation error %d\n",errno);
     result = s;
   }
 return result;
}
```

Debugging Tools

Debugging tools are provided in the infrastructure add-on packages:

- axp-ssh-4.6p1-k9.<platform>.<version>.pkg
- axp-app-dev.<platform>.<version>.pkg

The debugging tools are described in the following sections.

- SSH Tunneling, page 102
- Rsync, page 105
- GDB Debug, page 106

Note To return to the Table of Contents, click here.

SSH Tunneling

In a debugging/development environment, Cisco AXP provides access to the SSH server with full shell access. A user can login to a virtual instance as root.

Production Environment

In the production environment, for a tunnel_user SSH user, use the package axp-ssh-4.6p1-k9.<platform>.<version>.pkg.

The package does not have shell access on SSH login. You, as the developer, control what the tunnel_user can do by writing a startup script that is executed after SSH login.

Development Environment

In the development environment, for a tunnel_root SSH user, use the package axp-app-dev.<platform>.<version>.pkg.

The package provides shell access to the virtual instance upon SSH login. See the "Application Development Package" section on page 24.

Forwarding Examples

There are two SSH tunneling options for the virtual instance:

- Port Forwarding "ssh -L"
- X11 Forwarding "ssh -X"

The following manual pages for SSH describes these options in more detail:

http://www.die.net/doc/linux/man/man1/ssh.1.html.

Port Forwarding: Example

```
workstation-shell# ssh -L <local port>:<host>:<host port> <user>@<host> -p <port>
workstation-shell# ssh -L 7777:localhost:8888 tunnel_root@myblade -p 2022
```

X11 Forwarding: Example

```
workstation-shell# ssh -X <user>@<host> -p <port>
workstation-shell# ssh -X tunnel_root@myblade -p 2022
vserver-shell# xterm &
```

Set Up SSH Server in Virtual Instance

To setup a virtual instance to run an SSH server perform the following steps.

Note In the following steps, substitute the name of your application for <name>.

Step 1 Package the application with a dependency on the application development package axp-app-dev.<platform>.<version>.pkg

Step 2 Install the application and axp-app-dev.<platform>.<version>.pkg,

Step 3 Bind an interface to the virtual instance.

```
blade# configure terminal
(config)# app-service <name>
(config-myapp)# bind interface eth0
(config-myapp)# end
```

Step 4 Activate the user tunnel_root by setting a password.

```
blade# configure terminal
(config)# app-service <name>
(config-myapp)# ip ssh username tunnel_root password hello
```

```
(config-myapp)# end
```

Step 5 Activate the user tunnel_user by setting a password.

```
blade# configure terminal
(config)# app-service <name>
(config-myapp)# ip ssh username tunnel_user password hello
(config-myapp)# end
```

Step 6 Enable the virtual instance SSH server.

```
blade# config terminal
(config)# app-service <name>
(config-myapp)# ip ssh server
(config-myapp)# end
```

Step 7 Check that the virtual instance SSH server is running.

```
blade# app-service <name>
(myapp)# show ssh-server
Application SSH Server
Status:           RUNNING
```

Step 8 You can now connect to the virtual instance SSH server. The default port used is 2022.

The user connects in one of the following ways.

a. Obtain shell access by connecting with user tunnel_root as the SSH user.

```
workstation-shell# ssh tunnel_root@myblade -p 2022
tunnel_root@myblade's password:
vserver-shell#
```

b. Obtain access by connecting with user tunnel_user as the SSH user, via a startup script.

A message is displayed that depends on the startup script /usr/ssh/home/tunnel_user_app_startup.sh

For example, the example startup script shown below in tunnel_user_app_startup.sh: Example 1 displays message “Welcome Tunnel User!”. After the message is displayed, the remaining commands in the startup script are processed.

SSH Tunneling Examples

tunnel_user_app_startup.sh: Example 1

```
#!/bin/bash
echo "Welcome Tunnel User!"
echo "Please enter 1 to do a 'pwd', or 2 to do a 'ls'"
read choice
if [ $choice -eq 1 ]; then
   pwd
else
   ls
fi
```

tunnel_user_app_startup.sh: Example 2

Your script can invoke an SSH tunnel application directly:

```
#!/bin/bash
xterm
```

tunnel_user_app_startup.sh: Example 3

Your script can be interactive, allowing user a choice of invoking different commands.

```
#!/bin/bash
echo "Please enter 1 to start xterm, or 2 to start ethereal"
read choice
if [ $choice -eq 1 ]; then
xterm -a abc
else
ethereal -l abc -b -D
fi
```

Input/Output Using tunnel_user as the SSH user: Example

```
workstation-shell# ssh tunnel_user@myblade -p 2022
tunnel_user@myblade's password:
=====Start of message from the Cisco AppRE Application SSH Support=====
Tunnel User (tunnel_user) has logged in successfully
Application specific Tunnel User startup script will be invoked (if exists)
=====End of message from the Cisco AppRE Application SSH Support=======
Welcome Tunnel User!
Please enter 1 to do a 'pwd', or 2 to do a 'ls'
2
tunnel_user_app_startup.sh         tunnel_user_startup.sh
tunnel_user_app_startup.sh.sample
Connection to myblade closed.
workstation-shell#
```

Rsync

rsync is a utility for synchronizing files between the virtual instance on the service module and the development workstation.

During development you can move code from your development workstation to the virtual instance. Later, if you make changes to code on the virtual instance, you can synchronize with the code on the development workstation.

The *Cisco AXP User Guide* explains the prerequisite steps that must be completed before synchronization of files using the rsync utility. rsync is activated by using the **sync** command.

SUMMARY STEPS

1. **app-service** *application-name*
2. **sync file url rsync:***host_url* **direction [in|out] username** *username*
3. **exit**

Rsync: Example

```
exec-appname> sync file url rsync://10.1.1.1/source/container direction in username john
```

In this example, the content of the virtual instance is overwritten by the directory /source/container located at the developer workstation (address 10.1.1.1). `direction in` specifies that content from the workstation is used as the master file.

Note If you synchronize from workstation to service module while the virtual instance is running, some libraries such as libc may not work as expected. We recommend you do not synchronize while the system is running.

Updating System Libraries

To update system libraries on the service module, package the system libraries into your application and install the application.

If you need to update the libc library, also update libraries related to libc and libraries that are dependent upon libc. This ensures that there are no library version mismatches at runtime.

For example, when libc is updated, also update associated libraries such as libpthread, libm, libdl, libresolv. To find out more about libraries related to libc, see the libc documentation: http://www.gnu.org/software/libc/.

GDB Debug

A remote debugging service is provided by "gdbserver" that allows a remote debugger client to connect to and control the execution of applications on the Cisco AXP service module. The connection is secured via an SSH tunnel using OpenSSH software. The SSH tunnel enables you to debug applications regardless of the location of the Cisco AXP service module, even through a firewall. SSH tunnel authentication uses password authentication.

Debugging service "gdbserver" supports applications written in C/C++. Remote debugging will not be supported for interpreted (non-compiled) applications written in scripted languages including Bash, Perl, Python, TCL, and PHP.

If you want to use remote Java debugging you must include a remote debugger such as jdb or compile remote debug stubs in your application. An SSH tunnel may still be used with third party application supplied remote debuggers to secure the debugging session.

For further information on using gdb and gdbserver using SSH tunneling, see http://www.cucy.net/lacp/archives/000024.html.

The GDB debug utility provides debugging via the gdbserver and gdb commands.

Connect via an SSH tunnel to the virtual instance.

To run "gdb", perform the following steps.

Step 1 Setup a debugging session on the application running in the virtual instance using gdbserver. See the "gdb Setup" section on page 106.

Step 2 Make a SSH tunnel connection ("-L" option for port forwarding, see the "SSH Tunneling" section on page 102 and an example in the "SSH Tunnel Connection" section on page 107).

Step 3 Run "gdb" on the workstation to connect to the SSH tunnel. See the "Running GDB on the Development Workstation" section on page 107.

gdb Setup

The following examples show two ways to start a GDB debugging session, setting up an ssh tunnel connection, and running gdb on the development workstation.

tunnel_root GDB Debug Session

Login to the virtual instance SSH server as “tunnel_root” and run “gdbserver” manually.

In this example, the application name is /my_app/debug

```
workstation-shell# ssh tunnel_root@myblade -p 2022
tunnel_root@myblade's password:
vserver-shell# gdbserver localhost:2222 /my_app/debug
Process /my_app/debug created; pid = 29165
Listening on port 2222
```

tunnel_user GDB Debug Session

Setup the tunnel_user startup script to do runtime gdbserver binding, using the application process ID. This requires the tunnel_user startup script to be changed, as shown in the following example code. In this example the application name is “debug”.

```
#!/bin/bash
pid=`ps -ef | grep debug | awk '{print $2}'`
echo "gdbserver will be running on process ${pid}, bound to port 2222"
gdbserver localhost:2222 --attach ${pid}
```

SSH Tunnel Connection

Assuming that port 2222 is bound to the gdbserver on the virtual instance side, the following example command establishes an SSH tunnel from workstation port 4567 to service module port 2222.

```
workstation-shell# ssh -L 4567:localhost:2222 tunnel_root@myblade -p 2022
```

Running GDB on the Development Workstation

This example assumes that port 4567 binds to the SSH tunnel on the workstation, as shown in the above example code in the “SSH Tunnel Connection” section on page 107.

```
workstation-shell-2# gdb
(gdb) file debug
(gdb) target remote localhost:4567
(gdb)
```

You can use typical GDB commands such as break, continue, and step.

Examples

This section includes the following example(s):

Note To return to the Table of Contents, click here.

GetRouterIP API: Example

Use the function GetRouterIP( ) to obtain the IP address of the Cisco IOS router. GetRouterIP( ) is located in library /lib/libipcserviceapi.so.

The following example specifies the include file of ipcServiceAPI.h in the following include statement:

```
#include <ipcServiceAPI.h>
```

C

```
unsigned int GetRouterIP( void);
```

Obtain the Cisco IOS router's IP address.

Parameters:

None

Returns:

4-byte value for IP address

Example:

```
#include <stdio.h>
#include <string.h>
#include <ipcServiceAPI.h>

#define maskA 0x000f
#define maskB 0x00f0
#define maskC 0x0f00
#define maskD 0xf000
main () {

    unsigned int ip = GetRouterIP();

    printf("Router ip address=%d.%d.%d.%d\n",
    (maskA & ip) , ((maskB & ip) >> 8), ((maskC & ip) >> 16),  ((maskD &
    ip) >> 24 ));
}
```

Net-snmp: Example

In this example, MIB information is obtained from the router.

The SNMP version 1 community string is used to access MIB information from the system description MIB node `".1.3.6.1.2.1.1.1.0"`.

The community string `demopublic` must be configured in the router.

In this example, `1.2.3.4` is the router IP address. `SNMP_MSG_GET` is used to retrieve information.

The following net-snmp APIs used in this example: snmp_sess_init, snmp_open, snmp_perror, snmp_log, snmp_pdu_create, snmp_add_null_var, snmp_synch_response, snmp_errstring, snmp_sess_perror, snmp_free_pdu, snmp_close.

```
// file snmpapp.c

#include <net-snmp/net-snmp-config.h>
#include <net-snmp/net-snmp-includes.h>
#include <string.h>

/* change the word "define" to "undef" to try the insecure SNMP version 1*/
#define DEMO_USE_SNMP_VERSION_3

#ifdef DEMO_USE_SNMP_VERSION_3
    const char *our_v3_passphrase = "The UCD Demo Password";
#endif

int main(int argc, char ** argv)
{
    struct snmp_session session, *ss;
    struct snmp_pdu *pdu;
    struct snmp_pdu *response;

    oid anOID[MAX_OID_LEN];
    size_t anOID_len = MAX_OID_LEN;

    struct variable_list *vars;
    int status;
    int count=1;

    /*
    * Initialize the SNMP library
    */
    init_snmp("snmpapp");

    /*
    * Initialize a "session" that defines who we're going to talk to
    */
    snmp_sess_init( &session );                   /* set up defaults */
    session.peername = strdup("1.2.3.4");

    /* set up the authentication parameters for talking to the server */

    /* set the SNMP version number */
    session.version = SNMP_VERSION_1;

    /* set the SNMPv1 community name used for authentication */
    session.community = "demopublic";
    session.community_len = strlen(session.community);

    /* Open the session */
    SOCK_STARTUP;
    ss = snmp_open(&session);                     /* establish the session */

    if (!ss) {
        snmp_perror("ack");
        snmp_log(LOG_ERR, "something horrible happened!!!\n");
        exit(2);
    }

    /*
    * Create the PDU for the data for our request.
    *   1) We're going to GET the system.sysDescr.0 node.
    */
    pdu = snmp_pdu_create(SNMP_MSG_GET);
    read_objid(".1.3.6.1.2.1.1.1.0", anOID, &anOID_len);
```

```
    snmp_add_null_var(pdu, anOID, anOID_len);

    /* Send the request*/
    status = snmp_synch_response(ss, pdu, &response);

    /* Process the response*/
    if (status == STAT_SUCCESS && response->errstat == SNMP_ERR_NOERROR) {
        /*
        * SUCCESS: Print the result variables
        */

    for(vars = response->variables; vars; vars = vars->next_variable)
        print_variable(vars->name, vars->name_length, vars);

    /* Manipulate the information */
    for(vars = response->variables; vars; vars = vars->next_variable) {
    if (vars->type == ASN_OCTET_STR) {
        char *sp = (char *)malloc(1 + vars->val_len);
        memcpy(sp, vars->val.string, vars->val_len);
        sp[vars->val_len] = '\0';
        printf("value #%d is a string: %s\n", count++, sp);
        free(sp);
    }
    else
        printf("value #%d is NOT a string! Ack!\n", count++);
    }
    } else {
    /* FAILURE: print error*/

    if (status == STAT_SUCCESS)
        fprintf(stderr, "Error in packet\nReason: %s\n",
        snmp_errstring(response->errstat));
    else
        snmp_sess_perror("snmpget", ss);

    }

    /* Clean up */
    if (response)
        snmp_free_pdu(response);
    snmp_close(ss);

    SOCK_CLEANUP;
    return (0);
} /* main() */
```

Remote Serial Device: Example

The following code example is an application to test a serial modem device.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <fcntl.h>
#include <termios.h>
#include <ctype.h>
#include <sys/time.h>
#include <stropts.h>
```

```
#include <sys/ioctl.h>

int serial_open(char *dev) {
    int ttyfd;

    if (dev == NULL) {
        printf("\nDevice Empty");
        return -1;
    }
         printf("Opening..%s",dev);
    ttyfd = open (dev, O_RDWR | O_NONBLOCK | O_NOCTTY | O_NDELAY);

    if (ttyfd < 0) {
        perror("Device Open Failed: ");
        return (-1);
    }
    fcntl(ttyfd, F_SETFL, 0);
    return ttyfd;
}

int device_send(int ttyfd, char* data, int len) {
    int n;

    n = write(ttyfd, data, len);
    if (n<0) {
        perror("Device send failed: ");
        return n;
    }

    printf("Send Worked: %d", n);
    return 0;
}

int device_recv(int ttyfd) {
    char data[128];
    int r=0;
    memset(data, 0, sizeof(data));

    r = read(ttyfd, data, 128);
    if (r<0) {
        perror("device_recv failed: ");
        return -1;
    }

    printf("\nReceived: %s, Len: %d", data, r);
    return 0;
}
int modem_read (int fd) {
    char buffer[255];  /* Input buffer */
    char *bufptr;      /* Current char in buffer */
    int  nbytes;       /* Number of bytes read */

    printf("\nReading...");
    /* read characters into our string buffer until we get a CR or NL */
    bufptr = buffer;
    while ((nbytes = read(fd, bufptr, buffer + sizeof(buffer) - bufptr - 1)) > 0)
    {
        bufptr += nbytes;
        if (bufptr[-1] == '\n' || bufptr[-1] == '\r')
            break;
    }
    if (nbytes<0) {
        perror("Modem Read Failed: ");
        return (-1);
```

```
    }

    printf("\nModem Replied: %s", buffer);
    return 0;
}

int select_data (int fd) {
    struct timeval timeout;
    int readsocks=0;
    fd_set socks;

    FD_ZERO(&socks);
    FD_SET(fd,&socks);
    timeout.tv_sec = 1;
    timeout.tv_usec = 0;

    while(1) {
        readsocks = select(FD_SETSIZE+1, &socks, (fd_set *)0,(fd_set *) 0, &timeout);

        if (readsocks < 0) {
            perror("select");
            exit(1);
        }
        if (readsocks == 0) {
            /* Nothing ready to read, just show that
               we're alive */
            //printf(".");
            continue;
        }
        if (FD_ISSET(fd,&socks)) {
            /* Data Avilable */
            if (modem_read(fd) < 0) {
                return -1;
            }

        }
    }

}
void raw_mode(int fd) {
    struct termios options;
    /* get the current options */
    tcgetattr(fd, &options);

    /* set raw input, 1 second timeout */
    options.c_cflag     |= (CLOCAL | CREAD);
    options.c_lflag     &= ~(ICANON | ECHO | ECHOE | ISIG);
    options.c_oflag     &= ~OPOST;
    options.c_cc[VMIN]  = 0;
    options.c_cc[VTIME] = 10;

    /* set the options */
    tcsetattr(fd, TCSANOW, &options);
}

int                  /* O - 0 = MODEM ok, -1 = MODEM bad */
init_modem(int fd)   /* I - Serial port file */
{
    char buffer[255];  /* Input buffer */
    char *bufptr;      /* Current char in buffer */
    int  nbytes;       /* Number of bytes read */
    int  tries;        /* Number of tries so far */

    for (tries = 0; tries < 3; tries ++)
```

```
    {
        /* send an AT command followed by a CR */
        if (write(fd, "AT\r", 3) < 3) {
            perror("Modem Write Failed: ");
            continue;
        }

        /* read characters into our string buffer until we get a CR or NL */
        bufptr = buffer;
        while ((nbytes = read(fd, bufptr, buffer + sizeof(buffer) - bufptr - 1)) > 0)
        {
            bufptr += nbytes;
            if (bufptr[-1] == '\n' || bufptr[-1] == '\r')
                break;
        }
        if (nbytes<0) {
            perror("Modem Read Failed: ");
        }

        printf("\nRead %d Bytes(%s)", nbytes, buffer);

        /* nul terminate the string and see if we got an OK response */
        *bufptr = '\0';

        if (strstr(buffer, "OK") != NULL) {
            printf("\nModem Initialized");
            return (0);
        }
    }
    printf("\nModem Initialization Failed");
    return (-1);
}

int dial(int fd, char* number) {
    char cmd[64];

    memset(cmd, 0, 64);
    sprintf(cmd, "ATD %s\r", number);
    printf("\n(%d)Dialing...%s", strlen(cmd), cmd);
    if (write(fd, cmd, strlen(cmd)) < 3) {
        perror("Modem Write Failed: ");
        return -1;
    }
    return 0;
}

void print_termios(struct termios *options) {
    printf("\nTERMIOS ");
    printf("\nSpeed: %d", cfgetispeed(options));
    printf("\n");

}

void set_ioctl_baud(int fd, speed_t speed)
{
    struct termios ttyset;

    if (ioctl (fd, TCGETA, &ttyset) < 0) {
        perror("IOCTL GETA ERROR");
        exit(1);
    }


    print_termios(&ttyset);
```

```
    ttyset.c_ispeed = speed;
    ttyset.c_ospeed = speed;

    if (ioctl (fd, TCSETA, &ttyset) < 0) {
        perror("IOCTL SETA ERROR");
        exit(1);
    }

}


int set_baud(int fd,  speed_t speed) {
    struct termios options;

    /*
     * Get the current options for the port...
     */

    tcgetattr(fd, &options);
    print_termios(&options);
    /*
     * Set the baud rates to 19200...
     */

    cfsetispeed(&options, speed);
    cfsetospeed(&options, speed);

    /*
     * Enable the receiver and set local mode...
     */

    options.c_cflag |= (CLOCAL | CREAD);

    /*
     * Set the new options for the port...
     */

    tcsetattr(fd, TCSANOW, &options);

    tcgetattr(fd, &options);
    print_termios(&options);
}

int main(int argc, char *argv[]) {

    char *dev;
    int r, ttyfd;
    if (argc > 1)
        dev = strdup(argv[1]);
        else
        dev = strdup("/dev/modem");
    ttyfd=serial_open(dev);
    if (ttyfd) {
            //raw_mode(ttyfd);
            //init_modem(ttyfd);
            set_ioctl_baud(ttyfd, B4800);
            //dial(ttyfd, "9809649");
            //select_data(ttyfd);
            close(ttyfd);
    }
    printf("\n");
}
```

RPM File Extractor Tool

The RPM File Extractor Tool extracts all the RPM files into the project source root directory.

From the project source root directory, you can examine any dependencies the RPM files require, and view any preinstall, postinstall, preuninstall, or postuninstall scripts that the RPM files contain.

For more information on the RPM file extractor tool, see the following sections:

- Running the RPM File Extractor Tool, page 115
- RPM File Extractor Tool Arguments, page 116
- RPM Troubleshooting, page 116

Note To return to the Table of Contents, click here.

Running the RPM File Extractor Tool

To use the RPM File Extractor Tool (rpm_extractor), perform the following steps.

Step 1 Create a project directory if you have not done so already. This directory can be the same directory where the packaging tool runs.

Step 2 Enter the following command and arguments:

```
workstation# rpm_extractor.sh --proj <Project Directory> [--output] [--scripts] [--deps]
<RPM files> [--help]
```

For details of the arguments see the "Appendix B: Commands in Guest OS" section on page 124.

The RPM File Extractor File tool creates a sub-directory structure under the <project directory> as shown below.

```
myprojdirectory/
    rpm_extractor/
        output/
        scripts/
        deps/
```

Step 3 After the RPM scripts have been extracted (either by using Standard RPM Utilities, page 44 or the RPM File Extractor Tool, page 45), look at the listed preinstall scripts, or postinstall scripts, and RPM dependencies.

Step 4 Determine which part of this extracted information is required and move it to the project source directory.

Note Cisco AXP only supports a postinstall script.

Step 5 Package your application and install it on the service module.

RPM File Extractor Tool Arguments

The arguments of the RPM file extractor tool are:

- `--proj <Project Directory>` : Designates where the project directory is located. Typically this is the same directory name as used in the packaging tool's **project-dir parameter**.
- `--output` : (Optional) Flag to specify if the `output/` directory is created; the `output/` directory contains extracted RPM files in the directory designated as the root (`/`) directory. For example, rpmfile1
- `--scripts` : (Optional) Flag to specify if the `scripts/` directory is created; `scripts/` directory contains scripts information. The script file name consists of the RPM filename suffixed with "`.scripts`".
- `--deps` : (Optional) flag to specify if the `deps/` directory is created; contains a dependency file for each RPM file with a name preceded by the name of the RPM file. For example, `rpmfile1.deps`. Each dependency file indicates any further files required by this RPM file.
- `<RPM files>` : List of RPM files to be extracted. The RPM files are extracted to the directory designated as the root directory(`/`). Use either relative or absolute path names.
- `--help` : Prints help information for the RPM file extractor tool.

Note If none of the three optional arguments to the RPM file extractor tool are specified, all three subdirectories are created: `output/ scripts/ deps/`. (The optional arguments are: `[--output] [--scripts] [--deps]`.)

RPM Troubleshooting

The first time your application containing RPM files is installed onto the service module, you may encounter problems such as missing libraries, or missing configurations.

- **Missing Libraries**

  If there are further libraries missing, locate the missing libraries and transfer them using ftp to the service module, and retest the application. This avoids having to repackage your application before running it on the service module.

- **Missing Configurations**

  Configuration scripts extracted from RPM files may not run successfully. A simple debugging step is to access the Linux shell of the application and manually invoke the scripts—to try and isolate the problem.

Appendix A: Porting an Application from Fedora Core 6: Examples

This section shows two examples of porting an application from a Fedora Core 6 development environment onto Cisco AXP.

Example 1 is a simple C++ application. Example 2 is a more complex application that ports httpd software from the Apache Software Foundation.

This section consists of the following stages:

Note To return to the Table of Contents, click here.

Setup

The development environment setup used is based on a Fedora Core 6 workstation.

The Fedora Core 6 workstation is used for both of the following:

- port source platform
- development machine

Set up the machine and prepare it for Cisco AXP development.

- Install the base OS on the workstation—Fedora Core 6.
- Install and configure the tftp, ftp server.
- Install other packages needed for Cisco AXP SDK.
- Install sharutils (uuencode utility).
- If the XML processing software xmlstarlet is not already present, install xmlstarlet.
- Download and extract the Cisco AXP SDK.
- Create a workspace directory in which to work.

Example 1: Simple C++ Application

When a C++ application is compiled with the Fedora Core 6 compiler (3.4.4), the produced binaries are dependent at a minimum on the following libraries (found using ldd):

```
libstdc++.so.6
libm.so.6
libgcc_s.so.1
libc.so.6
```

```
/lib/ld-linux.so.2
```

These libraries are provided by Cisco AXP guest OS so that the minimum set of libraries does not have to be bundled as part of all third party application.

Ensure that the path LD_LIBRARY_PATH is setup prior to launching an application.

Example:

```
LD_LIBRARY_PATH=/usr/lib:/lib
```

For this simple application, there is no need to patch the libc. The application works with the Cisco AXP guest OS libraries. However, applications that are compiled in Fedora Core 6 have a different linking format, and when they are run in another environment they may fail with a floating point exception error. To alleviate this exception: compile your C/C++ application in Fedora Core 6 using either linking option "`-Xlinker -hash-style=sysv`" or "`-Xlinker -hash-style=both`".

When running your simple C++ application on the Cisco AXP blade, if you encountered a Thread Local Storage (TLS) error, then your program requires TLS support. Cisco AXP does not support TLS. However you can still port your application to Cisco AXP by including the libc libraries, including all libraries that provide support for libc.

See the "Updating System Libraries" section on page 106.

Example 2: Porting an Existing Application

The source operating system is Fedora Core 6 Linux—a popular distribution offering a large number of downloadable applications (RPM packages). To port Apache's httpd application to the network-module on Cisco AX, perform the following steps.

Download the RPM

Download the httpd RPM package from the Fedora Core website (compiled version, not the source package).

Note Clicking on the following link starts a download of the RPM.

http://download.fedora.redhat.com/pub/fedora/linux/core/6/i386/os/Fedora/RPMS/httpd-2.2.3-5.i386.rpm

Ensure that the package is for the i386 platform as in this example you will not rebuild the executable from source.

Extract the RPM

Extract the content of the RPM. This enables you to examine the content, modify the RPM file content as appropriate, add other files, and repackage.

For extracting the RPM use the provided Cisco AXP RPM Extraction tool. The RPM extractor tool is part of Cisco AXP SDK, located in the tools directory. To use the RPM extractor tool all you need is to create a directory and specify the extracted RPM file is to be written to this directory. Create a directory called extracted_httpd_rpm.

An example of extracting the RPM is:

```
../tools/rpm_extractor.sh --proj extracted_httpd_rpm httpd-2.2.3-5.i386.rpm
```

The rpm_extractor tool creates the following directory structure under the extracted_httpd_rpm directory:

```
rpm_extractor
deps
output
scripts
```

The `deps` directory contains a file, httpd-2.2.3-5.i386.rpm.deps, which lists the various commands and libraries that the extracted RPM file is dependent upon. The output directory contains the directory and file contents of the extracted RPM file. Copy these directories to the source directory. Use the files to package the httpd data. The scripts directory have a file, httpd-2.2.3-5.i386.rpm.scripts, which contains any preinstall, postinstall and preuninstall scripts. The preinstall and postinstall scripts can be copied into your Cisco AXP postinstall script that you use for your application, but you must verify that these commands will run on the Cisco AXP blade. Some changes to these commands may be required. Cisco AXP does not support preuninstall scripts.

Inspect the RPM Contents

This example reuses as much of the RPM content. httpd will not be running on a Fedora Core 6 distribution and you must make sure that the dependencies of the executable from the RPM are resolved.

To resolve the dependencies, go into each script and executable provided by the RPM and make sure that the script includes all the dependent software such as binaries, libraries and a configuration file that is then packaged into the new Cisco AXP installed package.

Identifying Libraries to Bundle

See the "Bundling Libraries in Your Application" section on page 123.

After running "strace" on the httpd executable, we identified the libraries that were loaded and copy them in our directory structure. It is important to also recreate the links that points to those libraries in our directory structure. Here is the list of the libraries and the symbolic libraries to include.

Other libraries that are libc-related libraries and are still required, but are covered in a later section:

From /lib:

libcom_err.so.2->libcom_err.so.2.1

libcrypto.so.0.9.8b

libcrypto.so.6->libcrypto.so.0.9.8b

libdb-4.3.so

libdistcache.so.1->libdistcache.so.1.0.1

libexpat.so.0->libexpat.so.0.5.0

libexpat.so.0.5.0
libgcc_s.so.1->libgcc_s-4.1.1-20061011.so.1
libnal.so.1->libnal.so.1.0.1
libpcre.so.0->libpcre.so.0.0.1
libselinux.so.1
libsepol.so.1
libssl.so.4->libssl.so.0.9.8b
libssl.so.6->libssl.so.0.9.8b
libuuid.so.1->libuuid.so.1.2
From /usr/lib:
libapr-1.so.0->libapr-1.so.0.2.7
libaprutil-1.so.0->libaprutil-1.so.0.2.7
libaspell.so.15->libaspell.so.15.1.3
libbz2.so->libbz2.so.1.0.3
libbz2.so.1->libbz2.so.1.0.3
libcurl.so->libcurl.so.3.0.0
libcurl.so.3->libcurl.so.3.0.0
libgmp.so->libgmp.so.3.3.3
libgmp.so.3->libgmp.so.3.3.3
libgssapi_krb5.so->libgssapi_krb5.so.2.2
libgssapi_krb5.so.2->libgssapi_krb5.so.2.2
libidn.so->libidn.so.11.5.19
libidn.so.11->libidn.so.11.5.19
libk5crypto.so->libk5crypto.so.3.0
libk5crypto.so.3->libk5crypto.so.3.0
libkrb5.so->libkrb5.so.3.2
libkrb5.so.3->libkrb5.so.3.2
libkrb5support.so->libkrb5support.so.0.1
libkrb5support.so.0->libkrb5support.so.0.1
liblber-2.3.so.0->liblber-2.3.so.0.2.15
liblber.so->liblber-2.3.so.0.2.15
libldap-2.3.so.0->libldap-2.3.so.0.2.15
libldap_r-2.3.so.0->libldap_r-2.3.so.0.2.15
libldap.so->libldap_r-2.3.so.0.2.15
libperl.so
libpq.so.4->libpq.so.4.1
libpspell.so.15->libpspell.so.15.1.3
libpython2.4.so->libpython2.4.so.1.0

librt.so.1->librt-2.5.so

libsasl2.so->libsasl2.so.2.0.22

libsasl2.so.2->libsasl2.so.2.0.22

libselinux.so.1->../../lib/libselinux.so.1

libsqlite3.so->libsqlite3.so.0.8.6

libsqlite3.so.0->libsqlite3.so.0.8.6

libstdc++.so.6->libstdc++.so.6.0.8

libutil.so.1->libutil-2.5.so

libxml2.so->libxml2.so.2.6.26

libxml2.so.2->libxml2.so.2.6.26

libz.so->libz.so.1.2.3

libz.so.1->libz.so.1.2.3

Postinstall Script

When extracting the RPM package, the extraction tool reports any installation related scripts bundled in the RPM. The RPM system supports a different hook to trigger an installation related script compared to Cisco AX. Cisco AXP only supports a postinstall script.

Go through all installation scripts that are bundled in the original RPM package and "port" them to a Cisco AXP postinstall script to make sure that they run in Cisco AXP and successfully run the setup steps. These installation steps are specific to the needs of the package.

For example, installation steps can include: creating users, setting permission, creating directories, and setting up the base configuration. In the postinstall script, mimic the functionality of each of the original scripts and test the postinstall script in the Cisco AXP environment.

In the scripts directory created by the rpm_extractor tool, there is a file `httpd-2.2.3-5.i386.rpm.scripts`. This file contains scripts extracted from the RPM file.

You must test each script manually on Cisco AXP to check that the commands are supported. Make any code changes required, and repackage the files using Cisco AXP SDK packaging tool.

Script Examples

The following examples are for preinstall, postinstall, and preuninstall scripts. (Cisco AXP does not support postuninstall scripts.)

preinstall scriptlet (using /bin/sh)

```
# Add the "apache" user
/usr/sbin/useradd -c "Apache" -u 48 \
        -s /sbin/nologin -r -d /var/www apache 2> /dev/null || :
```

Note Testing the preinstall script shows that Cisco AXP does not support the creation of system accounts (-r option). Also, you do not need to redirect the stderr to the bit bucket.

postinstall scriptlet (using /bin/sh)

```
# Register the httpd service
/sbin/chkconfig --add httpd
```

Note Testing the postinstall script shows that Cisco AXP does not support the `chkconfig` command. Therefore, you will probably need to create your own start and kill symbolic links in the `/etc/rc.d/rc3` and `rc6` directory, and have the links point to the httpd startup script in folder `etc/rc.d/init.d`.

Apache expects a group named "apache" with an id of 48 to exist.

preuninstall scriptlet (using /bin/sh)

```
if [ $1 = 0 ]; then
        /sbin/service httpd stop > /dev/null 2>&1
        /sbin/chkconfig --del httpd
fi

/bin/postinstall.sh

#!/bin/bash

#add group
/usr/bin/groupadd -g 48 apache

#add user
/usr/bin/useradd -c "Apache" -u 48 -s /sbin/nologin -d /var/www apache
```

Troubleshooting: Missing Configuration File

It is possible that when you start httpd on the network-module, httpd complains about a missing "`/etc/mime.types`" file not being present. The file comes from "`mailcap`" package which is not a dependent package. For this example, include the file from the "`mailcap`" package.

Identifying Cisco AXP Package Contents

Libraries

To avoid recompiling the httpd software from source code, include the Fedora Core 6 system libraries. This is required because the RPM executables were compiled against the Fedora system libraries and not the libraries in the virtual instance.

Fedora Core 6 comes with libc-2.5 and NPTL (Native POSIX Thread Library) support whereas the vserver provides libc-2.3.5 which is compiled *without* NPTL support. To solve this problem, all the libraries that are required by the httpd application must be included in the installation package.

Identify what is required to be copied from the workstation (running FC6) when installing the package. Taking all of the libraries would include unnecessary libraries taking up an unnecessarily large amount of disk space.

To identify the required libraries, you can use:

- ldd checker
- pkg_check.sh

These will tell you which libraries are missing in the Cisco AXP installation package and are not provided by the virtual instance.

Note Some libraries might not be identified as missing using the above checks, but these libraries should still be included in the installation package as well as being on the vserver. This is so that the library in the package is compatible with libc-2.5 from Fedora Core 6—the library in the package is used in preference to the library provided by the vserver.

You can identify the required libraries of an executable using the "ldd" tool and run it on the executable and the libraries that it depends on. Another alternative is to install httpd on the workstation by installing the RPM and run httpd using the "strace" utility.

Bundling Libraries in Your Application

Identifying Libraries to Bundle

One important step is to identify the libraries that you have to package with your application. This is a critical step since your application will not run if libraries are missing after the application is installed on the network-module.

To identify libraries, use one of the following options:

1) Cisco AXP ldd-checker—this tool detects the missing libraries. It checks the application package and the list of libraries that are provided by the Guest OS. This is the preferred way to identify missing libraries.

2) ldd tool—you can also use the ldd tool directly.

3) Use the "strace" utility—this is a little different, as this tool is actually run on the blade and will let you know which libraries are loaded and where they are found on the system. This is convenient when troubleshooting library issues.

Libc files

If the binary software to be ported to Cisco AXP depends on a different version of libc (for example, Fedora Core 6 uses Libc 2.5), you will have to copy the Libc from the system and bundle it in your application. You will also need to copy the libraries that Libc are related to. If not, there will be some mismatch at runtime with the one that comes with the Cisco AXP guest OS.

Here is a list of soft links and the files they point to that should be included when bundling libc in an application:

ld.so.1 -> ld-2.5.so (Library loading)

libc.so.6 -> libc-2.5.so

libcrypt.so.1 -> libcrypt-2.5.so (Encryption)

libdl.so.2 -> libdl-2.5.so (Dynamic Linker)

libm.so.6 -> libm-2.5.so (Math library)

libnsl.so.1-> libnsl-2.5.so (Name Service Library)

libnss_dns.so.2 -> libnss_dns-2.5.so

libnss_files.so.2 -> libnss_files-2.5.so

libpthread.so.0 -> libpthread-0.10.so (POSIX Thread Support)

libresolv.so.2 -> libresolv-2.5.so (Name resolution)

librt.so.1 -> librt-2.5.so (Realtime Extension)

libthread_db.so.1 -> libthread-1.0.so (linuxthreads)

libutil.so.1 -> libutil-2.5.so (Utility)

** You need to make sure that all the software links related to those libraries provided by the Guest OS got overwritten so that everything is pointing to the packaged libraries.

Library location and LD_LIBRARY_PATH

Note Libraries are looked up based on the LD_LIBRARY_PATH environment variable. Make sure that it is correct.

If libraries are bundled inside the application under "/usr/lib" you must adjust the LD_LIBRARY_PATH to include the "/usr/lib" path.

Overwrite the Guest OS libraries by placing new libraries in "/lib" (not in "/usr/lib").

Appendix B: Commands in Guest OS

/bin/awk	/bin/env	/bin/netstat	/bin/sleep	# From /usr/bin
/bin/basename	/bin/expr	/bin/nice	/bin/snice	/usr/bin/cksum
/bin/bash	/bin/false	/bin/nohup	/bin/sort	/usr/bin/dig
/bin/cat	/bin/find	/bin/notty	/bin/strace	/usr/bin/expect
/bin/chgrp	/bin/free	/bin/pgrep	/bin/stty	/usr/bin/groupadd
/bin/chmod	/bin/getopt	/bin/pkill	/bin/tac	/usr/bin/groupdel
/bin/chown	/bin/grep	/bin/pmap	/bin/tail	/usr/bin/groupmod
/bin/cmp	/bin/gunzip	/bin/ps	/bin/tar	/usr/bin/groups
/bin/col	/bin/gzip	/bin/pwdx	/bin/telnet	/usr/bin/hdparm
/bin/colrm	/bin/head	/bin/python	/bin/tftp	/usr/bin/hexdump
/bin/cp	/bin/hostname	/bin/rename	/bin/top	/usr/bin/host
/bin/curl	/bin/id	/bin/renice	/bin/touch	/usr/bin/ip
/bin/date	/bin/ipcs	/bin/rm	/bin/tr	/usr/bin/kill
/bin/dd	/bin/kill	/bin/rmdir	/bin/true	/usr/bin/killall
/bin/df	/bin/ln	/bin/sed	/bin/tty	/usr/bin/open
/bin/dir	/bin/ls	/bin/setsid	/bin/uname	/usr/bin/passwd
/bin/dirname	/bin/md5sum	/bin/setterm	/bin/uptime	/usr/bin/ping
/bin/dmesg	/bin/mkdir	/bin/sh	/bin/vi	/ /usr/bin/scp
/bin/du	/bin/mktemp	/bin/sha1sum	/bin/vmstat	/usr/bin/sftp
/bin/echo	/bin/more	/bin/skill	/bin/wc	/usr/bin/sftp-server
/bin/egrep	/bin/mv		/bin/whoami	/usr/bin/ssh
			/bin/xargs	/usr/bin/sshd
				/usr/bin/syslog_ng
				/usr/bin/tc
				/usr/bin/tcpdump
				/usr/bin/traceroute
				/usr/bin/useradd
				/usr/bin/userdel
				/usr/bin/usermod
				/usr/bin/which

Appendix C: Libraries in Guest OS

```
/lib/ld-2.3.5.so
/lib/ld-linux.so.2
/lib/ld.so
/lib/libblkid.so
/lib/libblkid.so.1.0
/lib/libcom_err.so
/lib/libcom_err.so.2.1
/lib/libc-2.3.5.so
/lib/libc.so.6

/lib/libcares-1.2.0.so
/lib/libcares.so
/lib/libcares.so.1

/lib/libcrypt-2.3.5.so
/lib/libcrypt.so
/lib/libcrypt.so.1

/lib/libcrypto.so
/lib/libcrypto.so.0.9.8
/lib/libcrypto.so.0.9.8a
/lib/libdecrypt_lib.so
/lib/libdecrypt_lib.so.0.9.8
/lib/libdecrypt_lib.so.0.9.8
a
# Curl Related
/lib/libcurl-7.15.2.so
/lib/libcurl.so
#
# glibc-2.3.5 Related
#
/lib/libdl-2.3.5.so
/lib/libdl.so
/lib/libdl.so.2
#
/lib/libe2p.so
/lib/libe2p.so.2.3
#
# Expat
/lib/libexpat.so
/lib/libexpat.so.1
/lib/libexpat.so.1.95.8
#
# Expect
/lib/libexpect.so
/lib/libexpect.so.5.43.0
# ncurses
/lib/libform.so
/lib/libform.so.5
/lib/libform.so.5.4
/lib/libform.so
/lib/libform.so.5
/lib/libform.so.5.4
```

```
#
/lib/libpanel.so
/lib/libpanel.so.5
/lib/libpanel.so.5.4

# Bind related
/lib/libisc.so
/lib/libisc.so.8.6.1
#
/lib/libjavautil.so
/lib/libjavautil.so.1
/lib/libjavautil.so.1.0.0
#
/lib/libm-2.3.5.so
/lib/libm.so
/lib/libm.so.6
#
# ncurses
/lib/libmenu.so
/lib/libmenu.so.5
/lib/libmenu.so.5.4
/lib/libncurses.so
/lib/libncurses.so.5
/lib/libncurses.so.5.4
# Net-tools
/lib/libnet-tools-1.60.so
/lib/libnet-tools.so
/lib/libnsl-2.3.5.so
/lib/libnsl.so
/lib/libnsl.so.1
/lib/libnss_dns-2.3.5.so
/lib/libnss_dns.so
/lib/libnss_dns.so.2
/lib/libnss_files-2.3.5.so
/lib/libnss_files.so
/lib/libnss_files.so.2
# Libpcap
/lib/libpcap.so
/lib/libpcap.so.0
/lib/libpcap.so.0.9.3
# PCRE
/lib/libpcre.so
/lib/libpcre.so.0.0.1
/lib/libpcreposix.so
/lib/libpcreposix.so.0.0.0
# Procps
/lib/libproc-3.2.5.so
/lib/libproc.so
/lib/libpthread-0.10.so
/lib/libpthread.so
/lib/libpthread.so.0
/lib/libresolv-2.3.5.so
/lib/libresolv.so
/lib/libresolv.so.2
```

```
#
/lib/librt-2.3.5.so
/lib/librt.so
/lib/librt.so.1
/lib/libss.so
/lib/libss.so.2.0
/lib/libssl.so
/lib/libssl.so.0.9.8
/lib/libssl.so.0.9.8a
#  Tar utility
/lib/libtar.so
/lib/libtar.so.1.15.1
# Java Util udppacer
/lib/libudppacer.so
/lib/libudppacer.so.1
/lib/libudppacer.so.1.0.0
#
/lib/libutil-2.3.5.so
/lib/libutil.so
/lib/libutil.so.1
#
/lib/libuuid.so
/lib/libuuid.so.1.2
# Zlib
/lib/libzlib.so
/lib/libzlib.so.1.2.3
# mx4j-libs
/usr/lib/java/bcel-5.1.jar
/usr/lib/java/commons-loggin
g.jar
/usr/lib/java/jakarta-oro-2.
0.6.jar
# Log4j
/usr/lib/java/log4j.jar
# MX4J
/usr/lib/java/mx4j-impl.jar
/usr/lib/java/mx4j-jmx.jar
/usr/lib/java/mx4j-remote.ja
r
/usr/lib/java/mx4j-rimpl.jar
/usr/lib/java/mx4j-rjmx.jar
/usr/lib/java/mx4j-tools.jar
/usr/lib/java/mx4j.jar
# net-snmp 5.1.2 library
/lib/libsnmplib.so
/lib/libsnmplib.so.5
/lib/libsnmplib.so.5.1.2
```

Notices

The following notices pertain to this software license.

OpenSSL/Open SSL Project

This product includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit (http://www.openssl.org/).

This product includes cryptographic software written by Eric Young (eay@cryptsoft.com).

This product includes software written by Tim Hudson (tjh@cryptsoft.com).

License Issues

The OpenSSL toolkit stays under a dual license, i.e. both the conditions of the OpenSSL License and the original SSLeay license apply to the toolkit. See below for the actual license texts. Actually both licenses are BSD-style Open Source licenses. In case of any license issues related to OpenSSL please contact openssl-core@openssl.org.

OpenSSL License:

Copyright © 1998-2007 The OpenSSL Project. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions, and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. All advertising materials mentioning features or use of this software must display the following acknowledgment: “This product includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit (http://www.openssl.org/)”.
4. The names “OpenSSL Toolkit” and “OpenSSL Project” must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact openssl-core@openssl.org.
5. Products derived from this software may not be called “OpenSSL” nor may “OpenSSL” appear in their names without prior written permission of the OpenSSL Project.
6. Redistributions of any form whatsoever must retain the following acknowledgment:

   “This product includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit (http://www.openssl.org/)”.

THIS SOFTWARE IS PROVIDED BY THE OpenSSL PROJECT “AS IS”' AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE OpenSSL PROJECT OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT

LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

This product includes cryptographic software written by Eric Young (eay@cryptsoft.com). This product includes software written by Tim Hudson (tjh@cryptsoft.com).

Original SSLeay License:

Copyright © 1995-1998 Eric Young (eay@cryptsoft.com). All rights reserved.

This package is an SSL implementation written by Eric Young (eay@cryptsoft.com).

The implementation was written so as to conform with Netscapes SSL.

This library is free for commercial and non-commercial use as long as the following conditions are adhered to. The following conditions apply to all code found in this distribution, be it the RC4, RSA, lhash, DES, etc., code; not just the SSL code. The SSL documentation included with this distribution is covered by the same copyright terms except that the holder is Tim Hudson (tjh@cryptsoft.com).

Copyright remains Eric Young's, and as such any Copyright notices in the code are not to be removed. If this package is used in a product, Eric Young should be given attribution as the author of the parts of the library used. This can be in the form of a textual message at program startup or in documentation (online or textual) provided with the package.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. All advertising materials mentioning features or use of this software must display the following acknowledgement:

   "This product includes cryptographic software written by Eric Young (eay@cryptsoft.com)".

   The word 'cryptographic' can be left out if the routines from the library being used are not cryptography-related.
4. If you include any Windows specific code (or a derivative thereof) from the apps directory (application code) you must include an acknowledgement: "This product includes software written by Tim Hudson (tjh@cryptsoft.com)".

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

The license and distribution terms for any publicly available version or derivative of this code cannot be changed. i.e. this code cannot simply be copied and put under another distribution license [including the GNU Public License].

Cisco Trademarks