Cisco IOS Programmer's Guide/ Architecture Reference

Software Release 12.0
Fifth Edition
February 1999

Corporate Headquarters
Cisco Systems, Inc.
170 West Tasman Drive
San Jose, CA 95134-1706
USA
http://www.cisco.com
Tel: 408 526-4000
800 553-NETS (6387)
Fax: 408 526-4100

Text Part Number: 78-2051-05

Cisco IOS Programmer's Guide/Architecture Reference
Fifth Edition February 1999
Release 12.0

Fourth Edition December 1997, Release 11.3
Third Edition September 1996
Second Edition February 1996
First Edition July 1995

9801R

Writers/Reviewers: Scott Mackie, David Hampton, David Stine, David Katz, Rob Widmer, Bob Albrightson, Steven Lin, Bob Stewart, Kevin Herbert, Francis Bruneault, Paul Traina, Mani Bandi, Eric Decker, Srihari Ramachandra, Greg Stovall, Steve Preissman, Sandra Durham, Andrew McRae, Jenny Yuan, Aviva Garrett, Deborah G. Bennett, John Walker, Kelly Morse Johnson, William May, Tim Iverson, Ken Moberg, Kristen Marie Robins, Susan Purcell
Editors: Jane Phillips, Betsy Fitch
Production: Brenda DePaolis

Change History

Changes in the Fifth Edition (February 1999)

The following changes have been made in the fifth edition of this manual. These changes correspond to Cisco IOS Release 12.0.

Chapter	Change
Overview	New section: "Scalability Changes." (May 1998)
File System	New chapter. (June 1998)
Scheduler	Added the following new sections: "Important Coding Guidelines," "if_onesec Registry Removed," "Event Driven Route Adjustment Message," "API for Keepalive and Other Periodic Intervals." and "FYI: Backup System Changes." (June 1998)
Writing, Testing, and Publishing MIBs	Appended Cisco IOS Technical Note #2, *Testing and Publishing a MIB*, to the end of this chapter. Changed the title from "Writing MIBs" to "Writing, Testing, and Publishing MIBs." (May 1998)
Interfaces and Drivers	In the section, "Scalability Changes," added "Modular Interface Naming and Numbering." (December 1997) Incorporated Cisco IOS Technical Note #6, "*Using IDB Subblocks*" in the section "IDB Terminology" (August 1998). Incorporated Cisco IOS Technical Note #7, "*Subblock and VFT Infrastructure Changes*" in the section "Subblock and VFT Support in Release 12.0" (January 1999). FYI: See the new chapter, "Extensible Plugin Driver API" in *Cisco IOS Device Drivers: Fundamentals of Architecture and Code*. (August 1998)
Memory Management	New section, "Virtual Memory." (September 1998) Additions to table describing the Memory Pool Classes. (Novermber 1998)
System Initialization	New section: "Enhanced High System Availability (EHSA)." (September 1998)
Debugging and Error Logging	In the section "Configure the Cisco IOS Software to Generate a Core File," added information about two new commands. One writes core files to flash: **exception flash**. One saves information across reboots: **memory sanity**. (August 1998)
Command-Line Parser	In the section "Process "No" Commands," added information that commands must be explicitly added to the Config tree for the "no" prefix to be recognized. (September 1999)
Part 7: Other Useful Information	New part. Contains the 4 new chapters listed below. (October 1998)
Scalable Process Implementation	New chapter. Was Cisco IOS Technical Note #5. (October 1998)
Backup System	New chapter. (August 1998)
Verifying Cisco IOS Modular Images	New chapter. Was Cisco IOS Technical Note #4. (October 1998)
Writing DDTS Release-Note Enclosures	New chapter. Was Cisco IOS Technical Note #3. (October 1998)

Note The optional interrupt service routine (ISR) interface is not documented in this manual. See the functional specification, ENG-17683.

Changes in the Fourth Edition (December 1997)

The following changes have been made in the fourth edition of the *Cisco IOS Programmer' Guide/Architecture Reference*. These changes correspond to Cisco IOS Release 11.3.

Chapter	Change
Socket Interface	New chapter. (November 1996)
	Revised the table of functions for new and revised functions, and to add macros. For details on changes to APIs, see the preface to the *Cisco IOS API Reference.* (November 1997)
Timer Services	A paragraph was added at the end of the section "Operation of Managed Timers." (November 1996)
	A paragraph was added to the end of the section "Initialize Managed Timers." (November 1996)
	In the section "Example: Managed Timers," the code in the example was changed. (November 1996)
	A paragraph was added to the sections "Guidelines for Allocating Memory "on page15 and "Stop a Managed Timer "on page 1 2 . (November 1997)
Strings and Character Output	The %CC format descriptor was added to the section "Format Time Strings." (November 1996)
	In Table 16-4, a typographical error was fixed in the third row, first column. The correct format codes are TD and Td, not TD and Tc. (November 1996)
Writing, Testing, and Publishing MIBs	Updated RFCs to the current RFC numbers. (January 1997)
	In the"Textual Conventions" section, added new textual conventions. (January 1997)
	Rewrote the "Establish a New MIB" section, changing the procedure from CVS to ClearCase. (January 1997)
	Rewrote the "Compile a MIB" section. This section includes documentation for the new MIB compiler. (January 1997)
Writing Drivers That Interface with the Cisco I OS Software	Removed this obsolete chapter. See instead the new manual on the subject: "*Cisco IOS Device Drivers: Fundamentals of Architecture and Code.*"
Interfaces and Drivers	In the section "Scalability Changes," added information about using subblocks and lists to improve the scalability of features that need to access IDBs. (December 1997)
	In the section "Scalability Changes," documented the change of `MAX_INTERFACES` from a global to a per-platform value (`Maximum Interfaces Constant No Longer a Global Value`). Provided example of the new way to define a structure and allocate space. (December 1997)
Interprocess Communications (IPC) Services	Corrected and expanded the information in this chapter. Change bars indicate new or revised material. (December 1997)
Registries and Services	Made clarifications to the descriptions of the roles of the registry files.
	Added subsections on 11.3 compiler changes and their effects on the registry files.
	Added a section on the placement of *xxx*`_registry.o` in makefiles,
	Added a section on the 'show registry' support.
	(December 1997)
IF-MIB	New chapter. (December 1997)

Chapter	Change
Older Version of the Scheduler	Expanded information about the obsolete function `cfork()` to clarify how it set the priority of the process that was being created. (December 1997)
Writing, Testing, and Publishing MIBs	Changed this chapter from an appendix into a chapter, "Writing, Testing, and Publishing MIBs." (December 1997)
Part 6	New part. Includes the chapters "Command-Line Parser," "Writing, Testing, and Publishing MIBs," and "IF-MIB." (December 1997)
Title	Added the phrase "Architecture Reference" to the title: *Cisco IOS Programmer' Guide/Architecture Reference*. (December 1997)

Changes in the Third Edition (September 1996)

The following changes have been made in the third edition of the *Cisco Internetwork Operating System Programmer's Guide*. These changes correspond to Cisco IOS Release 11.2.

Chapter	Change
Scheduler	The section "Change the Value of a Watched Boolean" has been changed to the following: Set the Value of a Watched Boolean Setting the value of a managed boolean to `TRUE` moves all processes watching this variable onto their appropriate processor ready queue if they are not already there. To do this, use the `process_set_boolean()` function: `void process_set_boolean(watched_boolean *wb, boolean value);` Once the process has been processed, the value of the managed boolean should be set back to `FALSE`.
Memory Management	In the section "List of Region Classes," the `REGION_CLASS_PCIMEM` region class has been added. This class is for PCI bus memory, which visible to all devices on the PCI buses in a platform. It is an optional class.
	In the section "List of Memory Pool Classes," the `MEMPOOL_CLASS_PCIMEM` memory class has been added. This class is for PCI memory, which is present on some platforms. It is an optional class.

	The section "Lock and Return Memory" has been added:
	When there are multiple users of a block of memory (such as multiple processes), it often becomes necessary to lock a block so that it is not freed until every user has signalled that they are finished with it. Each block of memory has a reference count associated with it for this purpose. When a block is allocated, it has a reference count (or `refcount`) of 1. To increment the `refcount` for a block of memory, use the`mem_lock()` function:
	`void mem_lock(void *memory);`
	To attempt to return a block of memory, use the `free()` function. All allocated memory can be returned using `free()`.
	`void free(void *memory);`
	If `free()` is called with a block that has a `refcount` of 1, the block is returned to the memory pool from which it was created. If the `refcount` is greater than 1, `free()` decrements `refcount` and returns without doing anything further to the memory block. This mechanism allows any of the potential users of the memory block to be responsible for returning it without risking a memory leak. In this regard, `free()` is the logical equivalent of `mem_unlock()` when using locked blocks of memory.

	The section "Example: Lock Memory" has been added.

Chapter	Change
	The default threshold for the `MEMPOOL_CLASS_LOCAL` pool class in the section "Set the Low-Memory Threshold" was incorrect. The section was corrected to the following: The low-memory threshold is triggered when the amount of free memory in a pool drops below a specified amount. The default threshold for the `MEMPOOL_CLASS_LOCAL` memory pool class is 96 KB. Other memory pool classes have no default thresholds. To set or change the low-memory threshold, use the `mempool_set_fragment_threshold()` function: `void mempool_set_fragment_threshold(mempool_class class, ulong size);`
	The default threshold for the `MEMPOOL_CLASS_LOCAL` pool class in the section "Set the Fragment Threshold" was incorrect. The section was corrected to the following: The fragment threshold is triggered when the size of the largest block free in a memory pool is smaller than a specified amount. The default threshold for the `MEMPOOL_CLASS_LOCAL` memory pool class is 32 KB. Other memory pool classes have no default thresholds. To set or change the fragment threshold, call the `mempool_set_low_threshold()` function: `void mempool_set_low_threshold(mempool_class class, ulong size);`
	In the section "Create a Memory Chunk," a description of the chunk pool flags was added.
	The section "Lock a Memory Chunk" was added: If the chunk pool was created with the `CHUNK_FLAGS_LOCKABLE` flag set, every element in the chunk pool has a reference count associated with it that can be incremented by the `chunk_lock()` function: `boolean chunk_lock(chunk_type *chunk, void*element);` The locking of chunk elements with `chunk_lock()` is exactly analogous to the `mem_lock()` function for data blocks allocated via `malloc()`, and the same examples and warnings apply.
Platform-Specific Support	In the section "Platform-Specific Strings," the strings `PLATFORM_STRING_HARDWARE_REWORK` and `PLATFORM_STRING_LAST_RESET` have been added.
	In the section "Platform-Specific Values," the value `PLATFORM_VALUE_LOG_BUFFER_SIZE` has been added.
ANSI C Library	New chapter. The "ANSI C Library" chapter in the Cisco IOS API Reference describes the ANSI C library functions supported by the Cisco IOS software.
Subsystems	The subsystem class, `SUBSYS_CLASS_REGISTR` , has been added.
Registries and Services	Major portions of the registries sections have been rewritten. The `registry_create()` function has been added.
Timer Services	The text in the section "Stop a Managed Timer" has been modified to the following: To stop a managed timer, use the `mgd_timer_stop()` function. This function can be used for both leaf and parent timers. If the timer is a parent, this function recursively stops all the children of this parent. This is useful for such operations as shutting down a process, because it is not necessary to find all the running timers. This function can be called regardless of whether the timer is already running, and it can be called from interrupt routines. If the timer is not running, this function does nothing. `void mgd_timer_stop(mgd_timer *timer);` A stopped timer is completely unlinked from the managed timer tree, so it is safe to free the memory containing the timer
Debugging and Error Logging	The section "Example: Trace Buffer Leaks" was added.

Chapter	Change

Binary Trees	In the section "Initialize a Wrapped AVL Tree," the syntax of the `wavl_init()` function was corrected to the following:
	To initialize a WAVL tree, use the `wavl_init()` function. In this function, you pass the `wavl_handle` you want to initialize, the number of AVL trees you want under this wrapped AVL tree, and a comparison routing for each of the AVL trees. You must call this function before calling any other wrapped AVL function.
	`boolean wavl_init(wavl_handle *handle,`
	`void *(*findblock)(wavl_node_type *), ...);`

Writing Cisco IOS Code: Style Issues	New chapter.
CPU Profiling	New chapter.

Changes in the Second Edition (February 1996)

The following changes have been made in the second edition of the *Cisco Internetwork Operating System Programmer's Guide*. These changes correspond to Cisco IOS Release 11.1.

Chapter	Change
System Initialization	New chapter.
Scheduler	Descriptions of new process queueing functions were added in the "Enqueue Data for a Process" section. The "Manage Sets of Scheduler Objects" section was added to describe the new functions `process_pop_event_list()` and `process_push_event_list()`.
Pools, Buffers, and Particles	Chapter was renamed from "Buffer Management." Description of Particles and Particle Pools was added. Some functions were renamed.
Interfaces and Drivers	Added inline functions in the section "Manipulate IDB Subblocks." Added the section "Use IDB Helper Functions."
Interprocess Communications (IPC) Services	New chapter.
Subsystems	Added the section "Tips for Creating a Subsystem."
Timer Services	Added 64-bit timers.
Strings and Character Output	Updated the timestamp `print()` format codes to add support for 64-bit timers.
Debugging and Error Logging	New chapter.
Binary Trees	Expanded the section "AVL Trees." Added the section "Manipulate Radix Trees."
Switching	New chapter.
Writing Drivers That Interface with the Cisco IOS Software	New chapter.
Porting Cisco IOS Software to a New Platform	New chapter.
Writing, Testing, and Publishing MIBs	New chapter.
Cisco IOS Software Organization	New chapter.
Glossary	New chapter.

CONTENTS

Figures

Tables

About This Manual

This section discusses the objectives, audience, organization, and conventions of this publication.

Document Objectives

This publication provides an overview of the structure and design of the Cisco Internetwork Operating System (Cisco IOS) development code and it describes the tasks necessary to write code for the Cisco IOS code elements. It does not provide function syntax descriptions. Therefore you must use this publication in conjunction with the *Cisco IOS API Reference* publication. This publication has documented Releases 11.0, 11.1, 11.2, 11.3, and 12.0 of the Cisco IOS software.

This edition focuses on the changes in the current release, 12.0. In this release and the prior one, 11.3, emphasis was placed on enhancing the modularity and scalability of the IOS software to make it more flexible in accommodating frequent updates and many interfaces. In the first part of Chapte r1, "Overview," these enhancements are summarized.

Audience

This publication is intended for Cisco internal development and test engineers, including new engineers and engineers at companies acquired by Cisco, such as LightStream, Newport, and Kalpana. This publication is also intended for Cisco internal porting engineers, who are porting the Cisco IOS software to third-party hardware and software products.

Document Organization

This publication is divided into seven main parts. Each part comprises chapters describing related functions of the Cisco IOS software. The organization of parts and chapters in this publication matches the organization of parts and chapters in the *Cisco IOS API Reference*, except that the reference publication does not contain appendixes. The parts in this publication are as follows:

- Part 1, "Overview," gives an overview of the Cisco IOS code. The chapters in this part describe the following:
  - "Overview," which describes the Cisco IOS components and summarizes thescalability enhancements in Releases 11.3 and 12.0
  - "System Initialization," which describes the initialization tasks performed by the Cisco IOS software when a platform is powered on or reset

- Part 2, "Kernel Services," describes the services provided by the Cisco IOS kernel. The chapters in this part describe the following services:
  - "Scheduler," which creates ready queues for processes and schedules processes for execution
  - "Memory Management," which you use to define memory regions, memory pools, free lists, and add support for virtual memory (in Release 12.0 and after)
  - "Pools, Buffers, and Particles," which describes the buffer support for sending, receiving, and forwarding packets to and from network devices
  - "Interfaces and Drivers," which discusses the interface descriptor block (IDB), the structure that describes the hardware and software view of an interface. The corresponding chapter in the *Cisco IOS API Reference* manual contains reference pages (man pages) for the IDB. For a discussion of the Extensible Plugin Driver API, available in Release 12.0, see *Cisco IOS Device Drivers: Fundamentals of Architecture and Code*.
  - "Platform-Specific Support," which describes the programming interface and developer hooks for handling platform-specific initialization issues, and for obtaining platform-specific strings and values
  - "Socket Interface," which describes the Cisco IOS socket interface implementation
  - "Interprocess Communications (IPC) Services," which describes Cisco IOS IPC services
  - "ANSI C Library," which describes the ANSI C library functions supported by the Cisco IOS software
- Part 3, "Kernel-Support Services," describes Cisco IOS services that are provided in support of the basic kernel services. The chapters in this part describe the following kernel-support services:
  - "Subsystems," which describes how the Cisco IOS code is organized into hierarchical modules
  - "Registries and Services," which permit subsystems to install or register callback functions, discrete values, or process IDs for a service provided by the kernel or other modules
  - "Timer Services," which support periodic processes, timeouts, and delay measurements
  - "Time-of-Day Services," which describes the Cisco IOS software time-of-day clock
  - "Strings and Character Output," which describes how to print strings and debugging messages
  - "Exception Handling" for the Cisco IOS software
  - "Debugging and Error Logging," which describes the Cisco IOS software debugging mechanisms
- Part 4, "Network Services," describes network services provided by the Cisco IOS software. The chapters in this part describe the following network services:
  - "Binary Trees," which are used by the Cisco IOS software for storage and keyed retrieval data structures
  - "Queues and Lists," which allow you to manipulate linked lists of data structures
  - "Switching," which describes the various routing and switching designs in the Cisco IOS code
- Part 5, "Hardware-Specific Design," describes how to customize the Cisco IOS software for the specific software on which it runs:

  - "Porting Cisco IOS Software to a NewPl atform," which provides guidelines for writing Cisco IOS code that is portable to different types of CPUs

- Part 6, "Management Services," describes how to manage a Cisco IOS network:
  - "Command-Line Parser," which describes the interface between the user and the Cisco IOS kernel
  - "Writing, Testing, and Publishing MIBs," which provides an overview of SNMP and MIBs and describes a general procedure for establishing a new MIB, attempting in the process to answer questions commonly asked by MIB developers and to address mistakes commonly made by developers
  - "IF-MIB," which explains how to support subinterfaces in the interfaces group MIB-II
- Part 7, "Other Useful Information," contains chapters that describe information that does not "slot" into the other six parts but nonetheless is important:
  - "Scalable Process Implementation," which addresses shortcomings in the Cisco IOS kernel code that interfere with developing scalable processes and describes ways to avoid these shortcomings
  - "Backup System," which is an overview of the redundant network connectivity scheme, as modified for Release 12.0
  - "Verifying Cisco IOS Modular Images," which explains how to verify Cisco IOS source code modularity, a modular image being a collection of linked object files that contain no unresolved references
  - "Writing DDTS Release-Note Enclosures," which provides guidelines for writing the text in which you report a problem in DDTS to customers
- Several appendixes provide the supplemental information:
  - "Writing Cisco IOS Code: Style Issues," which documents how to write Cisco I OS code, addresses the issues and conventions of the Cisco IOS software group.
  - "Cisco IOS Software Organization," which lists the subsystems in the Cisco IOS Release 11.1 software
  - "CPU Profiling," which describes a low-overhead method of CPU profiling that allows you to determine what the CPU spends its time doing
  - "Older Version of the Scheduler," which describes the features and API functions that were peculiar to the scheduler prior to Release 11.0
  - "Glossary," which defines some terms related to the Cisco IOS software
  - Inde

Document Conventions

Software and hardware documentation uses the following conventions:

- The symbol ^ represents the Control key.

  For example, the key combinations ^D and Ctrl-D mean hold down the Control key while you press the D key. Keys are indicated in capitals, but are not case sensitive.

- A string is defined as a nonquoted set of characters.

  For example, when setting up a community string for SNMP to "public," do not use quotes around the string, or the string will include the quotation marks.

Command descriptions use these conventions:

- Vertical bars (|) separate alternative, mutually exclusive elements.
- Square brackets ([]) indicate optional elements.
- Braces ({}) indicate a required choice.
- Braces within square brackets ([{}]) indicate a required choice within an optional element.
- **Boldface** indicates commands and keywords that are entered literally as shown.
- *Italics* indicate arguments for which you supply values; in contexts that do not allow italics, arguments are enclosed in angle brackets (< >).

Function prototype and macro descriptions, and C language keywords and code use these conventions:

- Text is in `screen` font.
- *`Italics`* indicate parameters for which you supply values; in contexts that do not allow italics, arguments are enclosed in angle brackets (< >).

Command examples use these conventions:

- Examples that contain system prompts denote interactive sessions, indicating that the user enters commands at the prompt. The system prompt indicates the current command mode. For example, the prompt `router(config)#` indicates global configuration mode.
- Terminal sessions and information the system displays are in `screen` font.
- Information you enter is in **`boldface screen`** font.
- Nonprinting characters, such as passwords, are in angle brackets (< >).
- Default responses to system prompts are in square brackets ([ ]).
- Exclamation points (!) at the beginning of a line indicate a comment line. They are also displayed by the router for certain processes.

Note Means *reader take note*. Notes contain helpful suggestions or references to materials not contained in this manual.

Caution Means *reader be careful*. In this situation, you might do something that could result in equipment damage or loss of data.

PART 1

Overview

CHAPTER 1

Overview

Moreover...they that weave networks, shall be confounded. Isaiah 18:9

1.1 Cisco IOS Software Components

The facilities and services available to application programmers writing code for platforms running Cisco IOS internetworking software can be divided into the following broad areas:

- Kernel Support Services
- Network Services
- Hardware-Specific Design
- Network Service and Protocols
- Management Services

This chapter provides an overview of each component listed above after giving an overview of the scalability changes in Releases 11.3 and 12.0.

1.2 Scalability Changes

The following changes were made to increase the Cisco IOS software's scalability:

- Subblock and Lists
- Extensible Plugin Driver API
- Event-Driven Scheduling
- Other Scalability Changes

1.2.1 Subblock and Lists

With Release 11.3, the policy has been to add no new fields to the interface descriptor block (IDB), the data structure that anchors interface state information. Instead, when adding a new interface, or adding a feature or option for an interface, a subblock has been used.

The use of subblocks has reduced the frequency with which all current IDBs need to be traversed. This frequency has been further reduced through the use of private, subblock-specific lists. For a discussion of subblocks and lists, see "Subblocks and Private Lists" in Chapter 6.

1.2.2 Extensible Plugin Driver API

On the 7200, C3600, and VIP platforms, the drivers that reside on a port adapter can share resources by creating a *plugin driver.* Details are documented in the "Extensible Plugin Driver API" chapter of *Cisco IOS Device Drivers: Fundamentals of Architecture and Code.*

1.2.3 Event-Driven Scheduling

Forms of polling that were not required by large numbers of programs have been replaced with event-driven methodology. The replacements have increased performance in all but one case. They have changed the way that you set keepalive frames and other periodic intervals and could hav affected the way that you should handle route adjustment messages. See the section "Scalability Changes" in Chapter 3 , "Scheduler."

1.2.4 Other Scalability Changes

Change	Documentation
Modular Interface Naming and Numbering	Chapter 6, "Interfaces and Drivers"
Maximum Interfaces Constant No Longer a Global Value	Chapter 6, "Interfaces and Drivers"
Enhanced High System Availability (EHSA)	Chapter 2, "System Initialization"
Interrupt Service Routine (ISR) API	ENG-17683, "*Cisco IOS Interrupt Service Routing API for All Platforms*" (11/04/97)

1.3 Kernel Services

The Cisco IOS kernel provides support for lightweight processes, memory resource management, exception handlers, buffer management, and other low-level and key services for an embedded platform. Processes respond to commands entered at terminals and perform tasks for users.

1.3.1 Scheduler

The scheduler used by the Cisco IOS kernel is simple, event-driven, and nonpreemptive. A process in Cisco IOS terminology is roughly equivalent to a thread in other operating systems. Processes are run until they manually relinquish the processor. They can be one of four different priorities—low, medium, high or critical). The priority of a process affects how often the scheduler considers it for CPU time. Because the scheduler is nonpreemptive, badly behaved low-priority processes can prevent critical processes from running.

The Cisco IOS software provides a variety of primitives for signaling that a process should awaken, including semaphores, timers, signal flags, work queues, and simple booleans. These primitives can be used in any combination, and a process can have several possible sources of events.

Processes can be created and destroyed at any point in time. Each process has its own stack, the size of which is specified when the process is created.

1.3.2 Memory Management

The Cisco IOS kernel expects its applications to run in an unmanaged memory environment. There is no support for providing user- and kernel-space memory regions, and no memory management is performed on process context switches.

Each platform has a distinct memory map that can include platform-specific memory areas to improve that platform's performance. The Cisco IOS kernel needs to know about as much of the platform's memory map as possible. The memory map is provided to the Cisco IOS kernel by the creation of regions. These are simply blocks of memory defined by a start address, a size, and memory and usage attributes. The platform-independent code then can use regions to derive memory map information without needing to know about platform specifics.

Memory pool support is provided to allow heap management and allocation of memory. Because a platform can have a variety of memory areas—such as local, shared, and fast—the memory pool support allows a variety of pools to be created to supply users of these various memory areas. Memory pools can manage discrete and disjoint regions of memory, allowing efficient usage of sparse and limited memory maps. The memory pools allow standard `malloc()` and `free()` operations to occur on the designated areas of memory.

1.3.3 Pools, Buffers, and Particles

Support for the handling and management network datagrams is integral to the Cisco IOS kernel. A buffer in the Cisco IOS software consists of two blocks of memory: a header block with context on the contents of a buffer and the data block, which holds the actual frame data. For simplicity, buffers are assumed to have contiguous data areas. In addition, buffers come only in a fixed range of data areas sizes and are managed by buffer pools.

Buffer pools hold a free list of identically sized buffers, all with the same memory attributes. Two different types of buffer pools can exist in the system: public and private.

Public buffer pools are created by default and are based on general media MTU and datagram sizes. They contain buffers with data areas that range from 100 bytes to 18 kilobytes in length. These buffer pools are available for all applications to use. Applications can dynamically grow and shrink the number of buffers available from public buffer pools to accommodate the demands of buffer usage.

Private buffer pools can be created by network drivers to manage specialized buffers for an interface. This allows drivers with particular memory alignment or size constraints to be accommodated cleanly.

The Cisco IOS software provides specialized primitives to allow applications to copy, trim, and adjust buffers.

The majority of the Cisco IOS protocol and application code assumes that the frame data presented to it are contiguous. The default buffer managed by the pool code has a contiguous area of memory for the frame data. In order to permit drivers to support scatter-DMA, the Cisco IOS software also allows frame data to be composed of individual blocks called *particles*. The use of particles is currently the exception rather than the rule in the Cisco IOS code base; before using particles in any code, seek design advice from senior Cisco IOS engineers.

1.3.4 Interfaces and Drivers

One of the fundamental data structures in the Cisco IOS software is the interface descriptor block (IDB). The IDB is split into a hardware descriptor, which describes an interface's physical channel, and a software descriptor, which describes an end point to which packets should be routed. A hardware IDB always has one software IDB associated with it. Additional software IDBs can be associated with a hardware IDB. This allows support of features such as subinterfaces and DLCIs.

Private lists of IDBs allow router feature code to create and maintain a list of IDBs that is a list of only those IDBs that have the feature in question enabled. This allows searches of IDBs to scale efficiently as the number of interfaces for a platform grows.

Protocols and drivers can attach information to hardware and software IDBs using the Cisco IOS subblock support. This support allows the attachment of data structures to an IDB without the IDB data structures needing to have explicit reciprocal knowledge about the attachment. Because the IDB structures are used throughout the Cisco IOS software, this method of transparently attaching structures allows information to be associated with an interface without other protocols and drivers being affected by the addition.

1.3.5 Platform-Specific Support

Much of the Cisco IOS kernel is generic. However, platforms impose peculiarities on kernel components such as system initialization, memory pool definition, and user interface displays. To allow platforms to be supported with the minimal amount of change in the generic code, the Cisco IOS software provides a platform API for platforms to hook themselves into. This interface is used by the kernel to obtain information such as the platform and vendor names, and the serial number of the chassis. The platform API also provides the hooks that the platform support code uses to declare memory regions, memory pools, and TTY devices to the Cisco IOS kernel.

1.3.6 Socket Interface

This Cisco IOS socket interface implements a subset of the standard UNIX socket functions. The Cisco IOS functions perform identically or almost identically to their counterparts in the standard UNIX socket library.

1.3.7 Interprocess Communications (IPC) Services

The Cisco IOS Interprocess Communications (IPC) services provide a communication infrastructure so that modules in a distributed system can easily interact with each other.

1.3.8 ANSI C Librar

Starting with Release 11.2, the Cisco IOS software is formalizing its use of ANSI C library functions. The "ANSI C Library" chapter in the *Cisco IOS API Reference* describes these library functions.

1.4 Kernel Support Services

The Cisco IOS kernel provides features and facilities that allow applications to be added to the system image with minimal impact to other elements in the system. It also provides support for many commonly expected application services, such as `printf()` and string manipulation libraries.

1.4.1 Subsystems

Subsystems allow the full image that is run on a platform to be pieced together from a palette of building blocks, many of which may be optional. This allows images to be constructed by selecting the pieces that are required without having to worry about linker dependencies.

All subsystems are declared by a subsystem header. Each subsystem has one header that provides all the information about the subsystem. The kernel uses the information in the header to initialize and install the subsystem into the system image. Each subsystem header is identified by a 64-bit "magic" number. All subsystems headers reside in the data segment of an image and are located automatically via their magic number when the system starts.

The main components of the header are the following:

- An entry point, which is called to initialize the subsystem
- A subsystem class specification, which indicates roughly when the subsystem should be called during platform initialization
- A set of properties that dictate dependencies of the subsystem. A requirements dependency allows subsystems to declare other subsystems on which they absolutely depend. If those subsystems are not found, the subsystem is not started. A sequence property allows intraclass initialization order to be specified. For example, if subsystem A must start before subsystem B and they are both of the same subsystem class, this information is specified in B's sequence property.

1.4.2 Registries and Services

Registries are primarily used by the Cisco IOS kernel to manage function vector tables. However, rather than just managing the vector jump address, they also allow the calling sequence to be specified. This also allows the context of the function vector invocation to be abstracted. The intent is that, although the applications that add to the service hooks can change, the code invoking the registry service does not need to change.

A *service* is a particular instance of associated function vectors, processes, or values with a particular invocation characteristic. A *registry* is a collection of associated services. The characteristic of a service dictates the calling sequence used. A case service, for example, emulates a case statement in C and matches a function vector to a specified value. A loop service calls all the function vectors registered for a service upon invocation.

Applications usually register functions with the system services at run time, often via the relevant subsystem initialization routine. After that, services can be invoked by any section of code.

1.4.3 Timer Services and Time-of-Day Services

The Cisco IOS software provides timer services for timing simple periodic events, performing duration timing, and so on. Timers can track time to the limits of the system clock accuracy, which is currently 4 milliseconds.

The Cisco IOS timer services implement the application-level functions by manipulating timestamps through a set of basic system calls. Typically, when a timer is set to expire at some point in the future, the system calculates the *epoch* (that is, the absolute time) of the expiration, and then the value of the system clock is watched until the expiration epoch is reached.

There are two types of timers: passive timers and managed timers.

Passive timers act by checking the system clock and noting either the time as is or the time after adding a delay value. The noted value is then examined periodically, either by polling or when triggered by an event.

Managed timers are groups of timers that run together. They can be arranged hierarchically so that only the timer at the highest level needs to be tested for completion rather than having to test all individual timers.

The Cisco IOS software also provides a rich set of time-of-day services. It contains a software time-of-day clock that can be interrogated and manipulated in various ways. Time can be manipulated and displayed in three formats.

1.4.4 Strings and Character Output

The Cisco IOS software provides functions for printing strings and debugging messages. Some of these functions are identical in many ways to the ANSI C functions of the same name. However, minor changes have been made to them to support Cisco IOS software-specific needs.

1.4.5 Exception Handling

The Cisco IOS system provides a limited form of exception handling that can be used by processes. This exception handling was originally designed to provide an easy method for processes to catch hardware exceptions, but it has been extended to provide limited software signaling. In no way is the Cisco IOS exception handling intended as a general-purpose signaling mechanism, as there are simple message passing and IPC primitives provided.

1.4.6 Debugging and Error Logging

The Cisco IOS software provides several debugging mechanisms for development engineers and support personnel. These include core file generation, a simple ROM-based debugger, a client debugging stub for host-based debuggers, formatted output routines for high-level tracing, and compile-time options to include additional tracing and logging.

1.5 Network Service

The Cisco IOS software provides various features that support network services, including support for binary trees and for queues and lists.

1.5.1 Binary Trees

The Cisco IOS software provides several data structures and utilities in generic libraries that allow you to store and retrieve large amounts of information quickly based on a keyed lookup. The Cisco IOS software provides three implementations of binary trees: Red-Black (RB) trees, AVL trees, and radix trees.

The Cisco IOS implementation of RB trees is a threaded tree. That is, once you find a node using a keyed search of the data structure, the only operation necessary to find the next higher or lower node in key order in the data structure is to follow a linked list. A variation on the RB tree—called interval trees—is also implemented in the same library as the RB tree. Interval trees are used when the key for an entry has an attribute of *width* or *range*.

AVL trees, named for Adel'son-Vel'skii and Landis, are balanced search trees. Balance is maintained in an AVL tree by use of rotations; as many as $O(\log n)$ rotations may be required after an insertion in order to maintain the balance of the tree.

1.5.2 Queues and Lists

The Cisco IOS software provides a variety of functions for manipulating linked lists of data structures. These functions support singly linked lists (sometimes also called queues) and doubly linked lists.

The functions for singly linked lists allow items can be added and removed from any position in the list. Data items can be on one or more queues simultaneously.

The Cisco IOS software provides support for simple doubly linked lists and for the list manager, which is a fully developed set of functions for manipulating doubly linked lists. Using the list manager, you can place the same item on multiple data structures.

1.5.3 Switching

The Cisco IOS software supports four different classes of switching:

- Slow switching (also known as routing). This class of switching is present in all routers.
- Fast switching. This class of switching is present in all routers.
- Autonomous switching. This class of switching is present only in routers with a ciscoBus, CxBus, or CyBus controller.
- Silicon switching. This class of switching is present only in routers with an SSE card.

1.6 Hardware-Specific Design

Various portions of the Cisco IOS software must be customized for the specific hardware on which it runs.

1.6.1 Porting Cisco IOS Software to a NewPlat form

One advantage of programs written in the C language is that they can be ported to a wide range of platforms. However, software in C can be written so that it is not portable to other platforms. This is true of parts of the Cisco IOS code, which assume a certain type of microprocessor.

1.7 Network Service and Protocols

Figur e1-1 shows a high-level overview of the network services and protocols available to run on top of the Cisco IOS software base. The left side of the diagram shows many of the routed protocols and their related routing protocols. Although these routed protocols allow many internals hooks to their routing protocols, the interface between the protocols and the rest of the Cisco IOS system for frame transmission and reception is actually relatively straightforward.

To transmit frames, the protocols use the relevant encapsulation modules—shown on the right side of the diagram—to add a media encapsulation. The encapsulation is specified by either the interface driver or via a user interface command. After the outgoing frame has been successfully encapsulated,

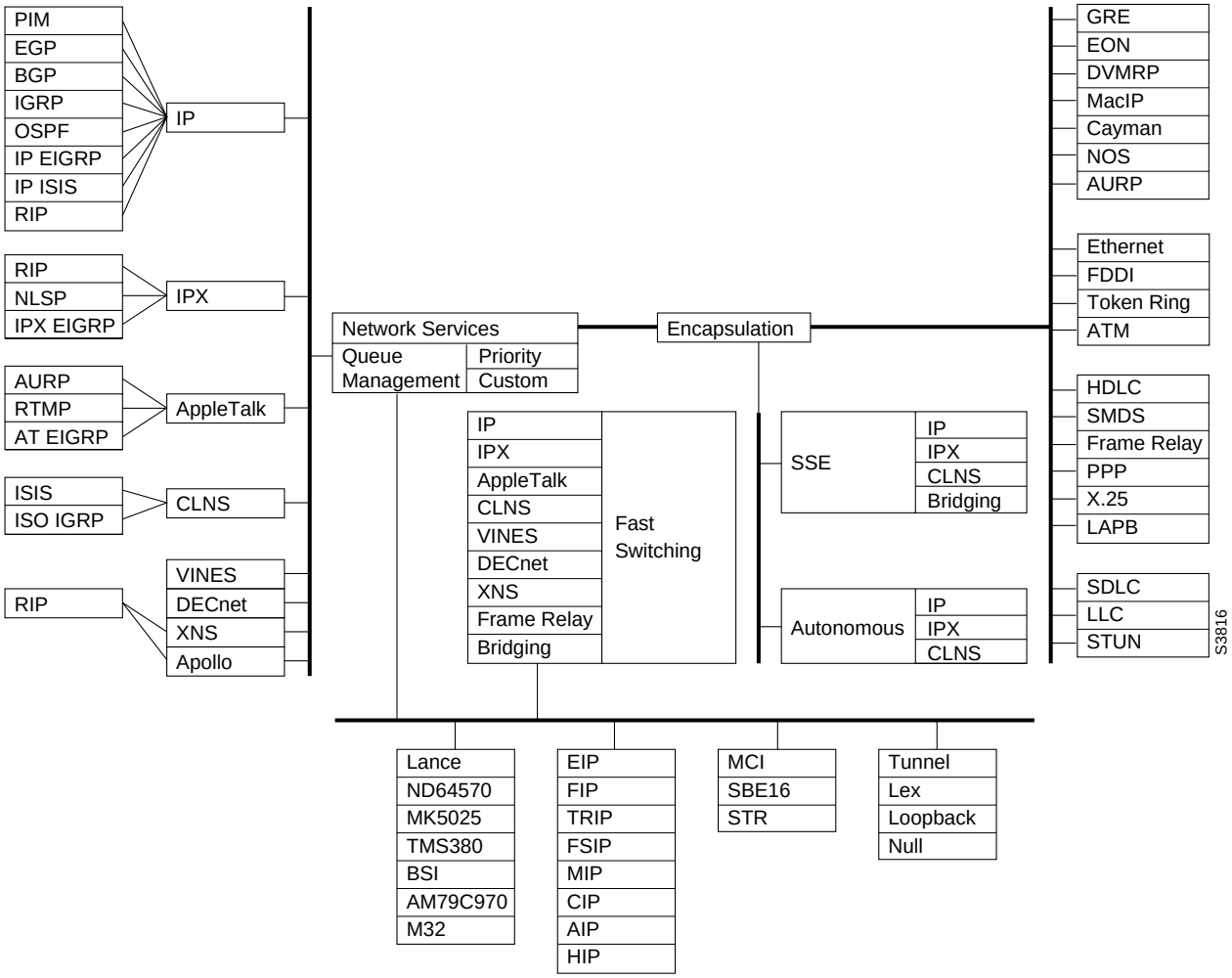
it can be enqueued on the outgoing interface queue for final transmission. The default enqueuing behavior for the outgoing queues is a simple FIFO, but if a specialized queuing algorithm is specified for the outgoing interface—such as priority or custom queuing—the frame is evaluated and enqueued using the rules for that queuing algorithm. The drivers—shown at the bottom of the figure—dequeue the next frame for transmission when they have space on their outgoing queues.

Frame reception is demultiplexed through a central incoming handler supplied with frames from the drivers. Each protocol registers a handler with the central demultiplexer and is called when a frame for that protocol is received.

Many protocols implement support for fast switching, which uses a cache of encapsulations that is built automatically from previous outgoing frames to quickly switch datagrams to an output interface. The fast-switching cache is populated by the encapsulation routines used by the protocols to send frames.

Autonomous and SSE switching are higher performance switching methods used by higher performance Cisco platforms. These switching methods also build various caches from the encapsulations of previous outgoing datagrams.

Figure 1-1 Cisco IOS Network Services and Protocols



1.8 Management Services

The Cisco IOS software provides facilities for network management from the command line and management applications.

1.8.1 Command-Line Parser

The Cisco IOS command-line parser is a finite state machine described by a series of macros that define the sequence of a command's tokens. Each macro defines a node in the state diagram of a command. This definition includes a pointer to the node to process if the current node matches the command-line input and an alternate node to process regardless of whether the current node is accepted. Optional parameters and keywords are indicated by alternate states in the parse tree. A macro exists for every type of object that can be parsed, such as keywords, integers, addresses, and string text.

1.8.2 Writing, Testing, and Publishing MIBs

The Simple Network Management Protocol (SNMP) supplies an interface for programming the management of network devices. It is the network-based access method to network devices used by Cisco's network management applications and those developed by third parties. "SNMP" is the commonly used name for the Internet-Standard Network Management Framework. This framework is a product of the Internet Engineering Task Force (IETF).

Cisco has supported SNMP version 1 (SNMPv1) for some years. As of Cisco IOS Release 10.2, Cisco implements a bilingual SNMP agent, which supports SNMPv1, SNMPv2. A primary component of SNMP is the Management Information Base (MIB), which defines data for observation and control and asynchronous notifications. Cisco implements several standard and enterprise MIBs.

Additional information about SNMP at Cisco is available at Cisco SNMP, and about MIBs at Cisco at SNMP MIB. If you are just starting to work with MIBs, off-the-shelf books are another good source of general information.

1.8.3 IF-MIB

With the addition of the IF-MIB (RFC 2233), the Cisco IOS software can now support subinterfaces in the interfaces group of MIB-II. To provide support for these new subinterfaces, you need to understand how to use the following components in registering or deregistering sublayers:

- Four key IF-MIB tables
  - `ifTable`
  - `ifXTable`
  - `ifStackTable`
  - `ifRcvAddressTable`
- Subiabtype data structure

  The `h/snmp_interface.h` file contains the subiabtype data structure. Be sure that you study and understand this structure. It is the one that you use to pass information across the IF-MIB API.
- Functions in the IF-MIB API.

A series of files holds the support for registering, updating, and deregistering subinterfaces in the `IF-MIB: snmp/ifmib_registry.reg, snmp/ifmibapi.[ch], h/snmp_interface.h,` and the `ifType` file.

CHAPTER 2

System Initialization

2.1 Overview: System Initialization

Each time a platform is powered on or reset, the Cisco IOS code performs a startup sequence of initialization tasks before the platform can begin routing and participating in network traffic transactions.

System initialization is divided into several sequential sections, each controlled by a particular piece of Cisco IOS code or a supporting item such as the ROM monitor. This chapter first describes the initialization sections, then describes the Enhanced High System Availability (EHSA) API, which was added in Release 12.0.

EHSA is a two-processor redundancy system: one takes over when the other dies or crashes. The EHSA section starts with an Overview that points out things to consider when designing EHSA support for your platform. It continues with an Implementation Guide that provides sample code and EHSA CLI syntax. The last section, EHSA Crash Handling gives a step-by-step description of what occurs when a processor crashes and specifies the information and functionality needed in platform code to handle a crash.

2.2 Basic Initialization

Basic initialization of a platform occurs after the platform is powered on or reset and before the Cisco IOS code initializes. Basic initialization consists of the following processes:

- Initialization by the ROM Monitor
- Bootstrap a Cisco IOS Image
- Allow the Cisco IOS Image to Take Control of the Platform
- Fundamental Initialization

2.2.1 Initialization by the ROM Monitor

Currently, each Cisco-proprietary platform running Cisco IOS software has a ROM monitor in it. The ROM monitor is responsible for initializing the platform so that it can launch the C iscoIOS software.

When a platform is powered on, the ROM monitor performs the following initialization steps:

1 Performs error checking for the hardware and verifies that the system is healthy.

2. Sizes and initializes memory to a known state. Initializing memory to a known state is required in order for parity to be enabled on systems where the DRAM is parity-checked for errors.

   Note that the ROM monitor does not initialize interfaces at this point, because it has knowledge only of the most basic hardware aspects of a platform.

3. What happens next depends on the settings of the configuration register. This register controls many aspects of platform bootstrapping. The actual form of the register varies from platform to platform. Many of the newer Cisco platforms store the configuration register settings in NVRAM; some older platforms use a DIP switch.

   If the configuration register is set not to autostart (bootstrap mode), the ROM monitor stops at its command-line interface and does not proceed further without user intervention.

   If the configuration register is set to autoboot, the ROM monitor continues and attempts to bootstrap a Cisco IOS image.

2.2.2 Bootstrap a Cisco IOS Image

How the ROM monitor attempts to bootstrap a Cisco IOS image depends on the platform. The following scenarios are used to bootstrap an image:

- Bootstrap a Cisco IOS Image from ROM
- Bootstrap a Cisco IOS Image from a Network
- Bootstrap a Cisco IOS Image from Flash Memory

2.2.2.1 Bootstrap a Cisco IOS Image from ROM

Many older Cisco platforms run the Cisco IOS image from ROM. On these platforms, both the ROM monitor and a full version of a Cisco IOS image are programmed into the same set of ROMs, which is then installed in the platform. For example, both the CSC/4 and the original IGS can boot thisway. When the platform is powered up, the ROM monitor dispatches directly into its companion copy of the Cisco IOS image—once the image is initialized—and runs the Cisco IOS image directly from ROM.

Bootstrapping a Cisco IOS image from ROM is the simplest form of booting. However, it is not a viable method for newer Cisco platforms because of the massive growth in image sizes and the serviceability aspects of upgrading code.

2.2.2.2 Bootstrap a Cisco IOS Image from a Network

A common method of booting a Cisco IOS image is to load a copy of the image over the network from a TFTP host. This is done by using an intermediate bootstrap.

The Cisco IOS software is structured so that it can act as a bootstrap to load another version of code. This allows the combination of the ROM monitor and a copy of a Cisco IOS image to load any other version of the Cisco IOS software.

The copy of the Cisco IOS software that is used as the bootstrap can come from one of two places:

- ROM—This bootstrap can either be a full version of a Cisco IOS image or an abridged version without routing called a *boot* image. The image resides in the same ROM set as the ROM monitor. Upgrading the bootstrap involves changing the ROM and can be laborious.

- Bootflash—This bootstrap is an abridged boot image in a separate bank of Flash called *bootflash*. Newer Cisco platforms use this bootstrap method to avoid having to manually upgrade the bootstrap so that it recognizes new interfaces. This method requires the newer ROM monitors, which use the BOOT variable to determine their bootstrap program.

Bootstrapping a Cisco IOS image from the network occurs as follows:

1. The platform requests bootstrapping from the network. This request can occur from two places:
   - ROM monitor command line. In this case, the command line is available to t he CiscoIOS software image for parsing.
   - System configuration (usually in NVRAM). In this case, the image name and host (if specified) are available from the configuration file.
2. The bootstrap attempts to load the Cisco IOS image onto the platform. The bootstrap grabs the remaining DRAM in the system and copies the image into it. Therefore, the bootstrap memory footprint should be kept as small as possible.
3. Once the image is loaded, the bootstrap terminates itself, returning control of the system to the ROM monitor.
4. If the image is compressed, the ROM monitor decompresses it.
5. The image is then relocated to its proper position in memory.
6. The system is restarted and executes the newly loaded copy of the Cisco IOS image.

2.2.2.3 Bootstrap a Cisco IOS Image from Flash Memory

Loading a Cisco IOS image from the network can be error-prone because of network and equipment outages. Many Cisco platforms provide support for Flash memory, which allows images to be copied and stored locally.

The sequence for booting from Flash memory is roughly the same as that for booting from the network, with the image being copied from local Flash memory rather than loaded over the network. However, there are a few differences. On newer platforms with bootflash, the ROM monitor must know how to access the Flash memory in order to load the bootstrap. On these platforms, full Cisco IOS images can be loaded directly from Flash memory without needing an intermediate bootstrap. On older platforms, a copy of the Cisco IOS software must be used to bootstrap the platform because the ROM monitor on these platforms knows nothing about the Flash memory present.

2.2.3 Allow the Cisco IOS Image to Take Control of the Platform

The entry point to the Cisco IOS image, which is usually called by the ROM monitor, allows the image to take control of the platform and begin executing. Each platform has a small section of code that handles the transition from ROM monitor to Cisco IOS image and is responsible for satisfying the platform-specific and image-specific needs of the system.

The following sequence of events occurs at the entry point to the Cisco IOS image:

1. What the code at the entry point does first depends on the platform on which the image is loaded.
   - For 680*x*0-based platforms, which have no MMU support, no initialization of a virtual to physical memory table is required.

- R4600-based platforms rely on the MIPS translation lookaside buffer (TLB) mechanism to build their address maps. As a consequence, the first thing that the R4600-based images do is construct the TLB table to use for that platform. (This table must be position independent.) Once this is done, normal addresses can be used and the execution path returns to the same one that the 680*x*0-based platforms use.

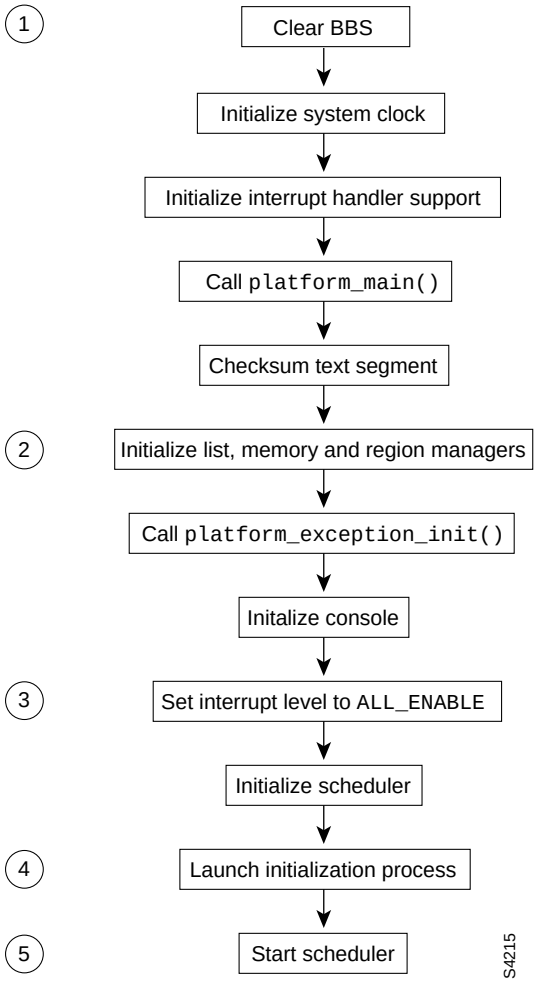
2. The generic code that follows the entry point first enables the basic GNU debugger (GDB) support for debugging.
3. What happens next depends on where the data segment for the image is located with respect to the text segment.
   - For images that are running from ROM or directly from a Flash bank, the data segment must be copied into DRAM so that it can be modified by a running system.
   - For images that are running from DRAM, no relocation is required.

Once bootstrapping for the Cisco IOS image is complete, the fundamental initialization of the Cisco IOS software image can proceed.

2.2.4 Fundamental Initialization

When a Cisco IOS image is launched, the initialization sequence illustrated in Figure2-1 is executed before the scheduler is started and processes begin to run.

Figure 2-1 Cisco IOS Fundamental Initialization Sequence



In Figur e2-1, there are five key points during the fundamental initialization sequence:

1 The BSS segment is cleared almost immediately at the start of the fundamental initialization sequence. (BSS is where the linker puts all global uninitialized variables.) Do not reference global data structures before this point unless you are extremely careful.

2 The memory management code is started comparatively late in the fundamental initialization sequence. No references to `malloc()` or `free()` can be made before this time. The `platform_memory_init()` function is called to allow platforms to declare their memory regions for the region and memory manager to control.

3 The ROM monitor always runs at the highest interrupt level possible. For example, on 680*x*0-based platforms, it runs at interrupt level 7. Up until this point, the initialization has run at the highest level, dropping to a lower level only to initialize the console because it was called directly from the ROM monitor. However, following this, all interrupts are effectively enabled. Therefore, all interrupt sources must either be squelched or fully handled by this time.

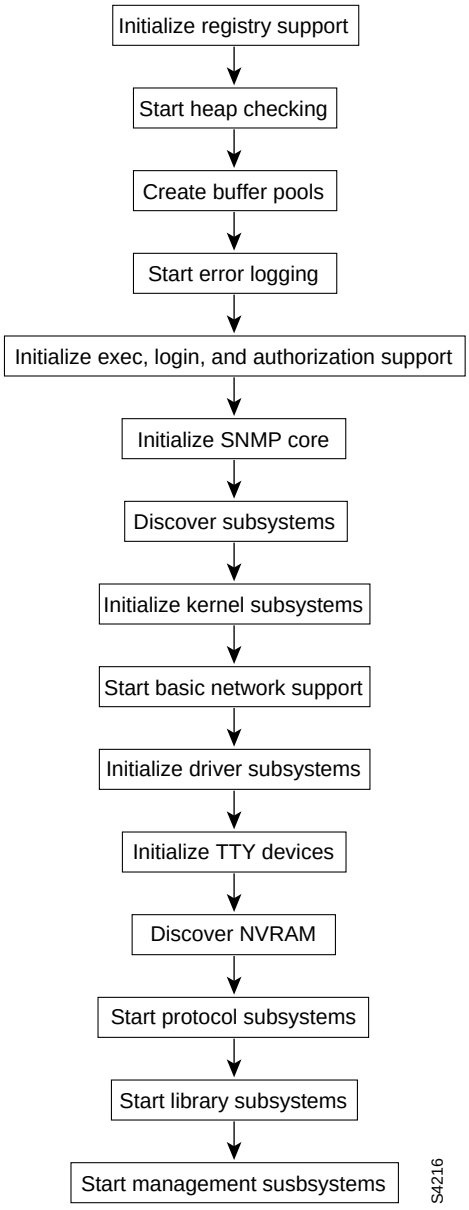
4 The remainder of Cisco IOS initialization is done from an initialization process. This process is created before the scheduler is running. Once the scheduler is started, the initialization process is run and the rest of the initialization completes.

5 The scheduler is started, and the initialization thread becomes the context from which the main loop of the scheduler is run. If the scheduler is stopped, the thread returns to the ROM monitor. This is how the reload mechanism returns control to the ROM monitor.

2.3 Cisco IOS Initialization Process

Most of the features associated with a Cisco IOS image are initialized from the initialization process created by the fundamental initialization context . Figure2-2 illustrates the system initialization sequence for the Cisco IOS kernel, network, and subsystem portions of the Cisco IOS code.

Figure 2-2 System Initialization Sequence



Once the initialization shown in Figure2-2 has been executed, the initialization process continues to execute a variety of boot and configuration options, such as loading the configuration from a TFTP server. Finally, the configuration is parsed to set the initial state of the interfaces and protocols. At this point, the initialization process terminates and the platform is considered initialized.

2.4 Enhanced High System Availability (EHSA)

EHSA is a redundant processor scheme: one processor takes over if the other happens to die or crash. This section describes how to implement EHSA, starting with an overview that spells out requirements and things to take into consideration when planning your EHSA strategy. Following the overview, an short implementation guide is provided, which shows you code samples from an implementation on two C7200 routers, refers you to specifications for other implementations, and delineates the framework for a cross-platform EHSA command line interface (CLI) syntax.

2.5 Overview

As Cisco moves into the carrier market, high availability (uptime) of an IOS image is becoming more and more critical. EHSA is an extension to the High System Availablity (HSA) feature that existed on the 7500 product. It is a platform-independent method of providing high availability.

EHSA works with two processors in the same chassis. These two processors might not share interface cards, or they might share some interfaces but not others.

Note Because EHSA is intended to be a two-processor implementationonly, if interfaces are not to be shared by the processors, the configuration and location of those interfaces should be identical and they should be connected e ternally with y cables.

EHSA, like HSA, has the two processors in a master-slave relationship, where the master is responsible for all IOS functionality. EHSA also has the slave processor do a of couple things: perform a 0 boot time initialization sequence and provide the framework for higher level protocols to provide master-to-slave updates.

2.5.1 Master-Slave Communications

EHSA will communicate from master to slave using Cisco IPC. Some platforms may need another form of communication to operate before IPC comes up.

Take care when updating IPC code to make your changes as backwards compatible as possible.

2.5.2 Health Monitoring

Health monitoring will be performed by an EHSA process. This process will be of critical priority. EHSA will send a keepalive from the master to the slave. The interval is determined by each platform and probably should be on the order of 1 or 2 times per second. The slave can use these keepalives, or the lack thereof, as one of the criteria for determining if it should take over.

The master should also have ways of informing the slave to take over immediately. These should be triggered through several different mechanisms. These include the **reload** command and a registry added to the crash routine, similar to the way a crash dump works. (See Ta ble2-1.)

The crash mechanism could also be used to alert protocols on the master that they should finish any communication with the slave, although this is a later phase project.

2.5.3 Slave Access and Information Requirements

The master processor needs to be able to access several devices on the slave. It needs to pass several pieces of information to the slave.

2.5.3.1 File System

The Route/Switch Processor (RSP) file system should be used on any platforms implementing EHSA. It currently contains IPC extensions that allow the master access to any device on the slave that the master is aware of. Right now, the master RSP can access the following slave devices:

- configuration (NVram)
- flash
- slot0
- slot1
- bootflash

The master does so by adding the prefix "slave" to each device and by using an *IPC-based remote filesystem* to each device. This remote file system over IPC has been made platform-independent.

The master has both a remote filesystem server and a remote filesystem client. Although the slave has a remote filesystem client, the remote file system server on the master is not currently used.

In addition, putting a remote filesystem client on the slave gives it the ability to access the master's tftpboot device, thus allowing the slave to netboot using the master's filesystem.

Any future enhancements to EHSA or the file system must guarantee the reliability of the file system service.

2.5.3.2 Boot Parameters

HSA passes certain parameter variables (`BOO` , `BOOTLDR`, and `CONFIG`) from the master to the slave. These should still be passed but have been changed to use environment-like variables.

2.5.3.3 Time

The master should update the real time clock on the slave. (This ability also exists for HSA.) There are two requirements:

1. System time

   System time is sent every minute or so to ensure that the slave's idea of time does not drift too much from the master's.

2. Battery-backed calendar

Battery-backed calendar time is sent to the slave whenever it has written the master, either because the user issued a command or because Network Time Protocol (NTP) is configured to periodically update the calendar.

The first is so that slave processes use the correct time, for example remote filesystem timestamps. The second is to ensure that after a changeover, the router boots with the correct time/date.

2.5.3.4 Future Projects

EHSA provides the framework for allowing the protocols and device drivers to pass state information. This includes infrastructure changes for allowing the protocol and driver subsystems to be informed if a slave is taking control from an active master, or if the processor is a master taking control for the first time (that is, a power on situation or one where there were two slaves vying for control).

This could be done by adding a new subsytem to register for when a slave comes up. A protocol or device driver could register an IPC session. Then, when the slave took over, it would be informed in the subsystem header that is passed as part of the PROTOCOL subsystem init.

2.5.3.5 Version Compatibility

Version compatibility is very important for passing information from the master to the slaves. This compatibility comes at several different layers.

The first layer is IPC. If an IPC connection cannot be established when the slave first comes up, there must be some way to determine who should be master. This should be on a platform-by-platform basis. Basic IPC must be compatible for all versions of EHSA.

The second layer of version compatibility is between the master and slave during the hello messages. If the slave is incompatible with the master, the slave should still come up but should never take over for the master. At a minimum, the remote file system code should always be available to copy new images to the slave.

The last layer of compatibility is for each of the services that uses EHSA. Each service needs to verify that it is compatible with the slave. This should occur each time a slave initializes contact with the master. For example, if a routing protocol implements a set of master-to-slave state machines, that protocol would be responsible for determining if a feature exists on the slave (or master) and reporting to the console if an incompatibility exists.

If a slave has more services than a master, this should not be cause for an automatic switchover. Operator intervention should be required to get the master and slave running more compatible versions.

Some services that must never become incompatible include keepalives and file system.

2.5.3.6 Auto Sync

Auto sync is a configuration copying service. When a configuration is saved to NVram (or **startup-config**), the master will copy that configuration to the slave.

Auto-sync should be defaulted on, and should run the **copy startup-config slavestartup-config** command.

ROM Monitor (rommon) variables and boot variables must also be auto sync'd.

Sync'ing of images is manual, and must be done by the operator.

2.5.3.7 Slave Console

While the slave is running in anticipation of being a master, several console commands should be accessible, including most **show** commands that access kernel information.

Certain commands will need to be filtered out, including allowing the slave to enter configuration mode, and all commands involving interfaces. Most commands will not be active in the system, since the driver and protocol registration has not taken place.

Some commands will have to be registered earlier.

The master should also have access to the slave's console. Something similar to the **if-console** command implemented for the VIP platform should be implemented. Only 1 **if-console** session should be active at any time.

It is desired to have GDB run over the **if_console** command. This is a requirement for platforms that have a single console for both processors. In that case, IPC might not be the correct transport mechanism for console messages.

We may also have to pass the console configuration from the master over to the slave, since we will not have run the initial NV configure on the slave.

2.5.3.8 Slave Message Logging on Master

The slave should be able to pass error messages to the master. The master should also be able to access the crash history of the slave. This is not a requirement but an option. It should be available on a per-platform basis.

2.5.3.9 Seamless Software Upgrades

A seamless software upgrade would be if the master loaded the slave with a new image, rebooted the slave, then switched from the master to the slave when the slave came up and was ready.

When the master was ready to switch over, it would inform all the services that are running on the master so that they could complete passing whatever information was current, then switch to the slave.

The ability to provide seamless software upgrade is part of EHSA. Implementation is dependent on the individual protocols and platforms.

2.5.3.10 Mac Addresses

This is a concern for platforms that do not share a common MAC address PROM on the backplane. In this case, the MAC addresses from one of the processors must be passed from the master to the slave.

On these boxes, to ensure that the MAC addresses remain the same, the MAC addresses should be written to the configuration. This is a platform-dependent item.

2.5.4 Basic Flow and Operation

2.5.4.1 Basic Slave Operation

To implement EHSA on a full image requires the slave to stay in a state before the interface driver class subsystems are initialized.

This should probably occur before, or at the time of, `platform_interface_init()`.

2.5.4.2 Initialization

EHSA can be broken down into the following initialization cases: a slave coming up with an existing master, both slaves coming up, and a slave coming up with no communication from the master.

When a slave is coming up with an existing master, this case should be determined quite rapidly because the IPC session should begin almost immediately and messages should be passed between the master and the slave. In this case, there should be no voting. There should be no way a slave that comes up should be allowed to become the master, unless the master crashes, or unless the operator orders the switchover.

The more difficult case is the second, when two slaves are coming up at the same time. In that case, IPC must come up and they must negotiate which is to be the master. We must come up with some generic mechanism (perhaps based on version number). If all things are equal, you need to provide a platform-dependent way to break the tie.

If, in the third case, IPC does not come up when a slave is initializing, the slave must wait a certain period before continuing with its initialization. This period should probably be a significant amount of time—30 seconds or so—to attempt to avoid any blackout periods that the other side might have during its own initialization, and should be done on a platform-to-platform basis. This does increase the time period of initialization for a complete failure.

Care should be taken during the initialization period not to disable interrupts, and to suspend so that the IPC task can run, and check if another slave has come up. This will require that the init process must periodically suspend.

2.5.4.3 Interaction with the Boot Loader Image

Most high-end routers contain a boot loader image, which consists of enough code to netboot or to load the real image from flash.

Boot loader images must also be aware of the master-slave relationship. They have the same problems with initialization that are discussed in the preceding section (Initialization).

Boot loader code only has a problem between master and slave if netbooting is tried, and only when there is not a clear master.

For information regarding a boot loader netbooting when a master is up, see File System above. For a boot loader to netboot when there is no clear master, one boot loader must become the master; therefore, EHSA code is required to be in the boot image. The slave must stay in the boot loader slave state until the master comes up fully, including not taking control from the time the master image finishes loading the image until the time the master gets to the negotiation point. This will require a different algorithm in the boot loader for EHSA slave takeover than in the production image.

2.6 Implementation Guide

This section shows you a sample implementation. The two C7200 routers in the sample are connected via a DEC21140 FE PA (port adapter). No other hardware support is implied. There is no ROM Monitor (rommon) support required.

Note Recommended reading in addition to this chapter: ENG-20467**,** *CPU Redundancy for Cougar.*

2.6.1 Initializing EHSA

The last opportunity for determining primary or secondary status and acting on that status is at the end of `platform_interface_init()`. This also appears to be the best place for initializing EHSA. At any rate, your platform must initialize EHSA before the end of `platform_interface_init()`.

2.6.1.1 SUBSYS_CLASS_EHSA

`SUBSYS_CLASS_EHSA` is for initializing subsystems that would normally be initialized in the driver subsystem but need to be initialized earlier for EHSA:

```
/*
+      * Initialize the EHSA(redundancy) subsystems.
+      */
+     subsys_init_class(SUBSYS_CLASS_EHSA);
```

2.6.2 EHSA APIs

Table 2-1 lists the EHSA API and registry functions, with a short description of each call. You will find reference pages on each of the API functions in the "System Initialization" chapter of the *Cisco API Reference* manual.

Table 2-1 EHSA Functions

API Call	Description
ehsa_event()	Tells EHSA to switch state.
ehsa_get_state()	Gets the current EHSA state.
ehsa_suspend_init()	Suspends the init process.
ehsa_resume_init()	Causes the init process to resume.
ehsa_secondary()	Become an EHSA Secondary: suspends the init process and starts the EHSA Secondary background process.
ehsa_primary()	Starts the Primary background process.
ehsa_init_control()	Initialize the control structure, ehsa_control_t.
Registry Call	**Description**
ehsa_switch_to_primary(void)	A list registry for functions that need to be run when a Secondary transitions to Primary. Invoked from `ehsa_event()`.
ehsa_switch_to_secondary(void)	A list registry for functions that need to run when a Primary dies. This should only be used for clean up prior to **crashdump** or **reload**. Invoked from `ehsa_event()`.
ehsa_switch_to_standalone(ehsa_event_t old_state)	A list registry for functions that need to run when either a Primary or Secondary becomes the sole controlling CPU in the chassis. The previous state is passed to the invoked function(s). Invoked from `ehsa_event()`.

2.0.0.1 Actions on Status -State Transitions

Table 2-2 EHSA State Meanings

State and Meaning	Trigger Event	Actions	Valid Transitions
`EHSA_STANDALONE` No other EHSA-capable card is currently known. It means that this CPU is the controlling CPU and is acting as a standalone. This is the initial state (when bss is initialized to 0 in `main()`). This is also the state that should be set when the hardware detects that the other EHSA card has been pulled from the chassis.	`ehsa_event(EHSA_SW_SA)`	`reg_invoke_ehsa_switch_to_standalone()` is invoked. If a CPU was previously a Primary, the EHSA Primary background process is killed. This stops all EHSA activity. EHSA must be re-activated if a EHSA-capable card is inserted. If a Secondary, this causes the EHSA code to resume the init process, kill the EHSA Secondary background process and cause `ehsa_secondary()` to return. This CPU will finish init and control the box.	`EHSA_PRIMARY`, `EHSA_SECONDARY`
`EHSA_PRIMARY` Two EHSA-capable cards exist and this one is Primary.	`ehsa_event(EHSA_SW_P)`	None. See `ehsa_primary()` in the API section above.	`EHSA_STANDALONE`, `EHSA_DEAD`
`EHSA_SECONDARY` Two EHSA-capable cards exist and this one is Secondary	`ehsa_event(EHSA_SW_SD)`	None. See `ehsa_secondary()` in the API section above.	
`EHSA_DEAD` This CPU is dead or dying. Last EHSA state before returning to ROM Monitor (rommon).	`ehsa_event(EHSA_SW_D)`	An attempt is made to kill any EHSA process and notify the other CPU that this CPU is dead. Best effort attempt. Type of error or crash may prevent this CPU from doing any of the above.	

2.0.0.2 Using ehsa_event() to Trigger State Transitions

When it detects a status change, of either the Primary or the Secondary, the platform must notify EHSA via the `ehsa_event()` call:

- If on the Secondary:
  - If the Primary has died or any other condition exists that prevents it from acting as Primary, use `ehsa_event(EHSA_SW_P)` (reinit as IPC master in the platform code). The EHSA code cleans up and then transitions to state `EHSA_PRIMAR` .
- If on the Primary:
  - If the Secondary has died, use `ehsa_event(EHSA_SW_SA)`.
  - If it self-detects an error, use `crashdump()`. EHSA crash handling will be invoked from `crashdump()`.

- If OIR:
  - If a new EHSA-capable card is being inserted, use `ehsa_init_control()`, `ehsa_event(EHSA_SW_P)` for the existing card, and `ehsa_event(EHSA_SW_SD)` for the new card.
  - If an existing Primary or Secondary is being removed, use `ehsa_event(EHSA_SW_SA)`.

2.0.1 Examples

The following sample code is from `platform_interface_init()` in `src-4k-c7100/platform_c7100.c`:

```
ehsa_discover(ehsa_hwidb);
        state = ehsa_get_state();
        switch (state) {
            case EHSA_SECONDARY:
                state = EHSA_PRIMARY;
                ehsa_slave_ipc_init(ehsa_info->hwidb, ehsa_info->other_mac);
                ehsa_secondary();
                break;
            case EHSA_PRIMARY:
                ehsa_master_ipc_init(ehsa_info->hwidb, ehsa_info->other_mac);
                ehsa_primary();
                break;
            case EHSA_STANDALONE:
                break;
            default:
                crashdump(0);
        }
```

In the example above, `ehsa_discover()` exchanges packets to determine role and calls `ehsa_event()` with one of the following: EHSA_SW_SA, EHSA_SW_SD, EHSA_SW_P, EHSA_SW_D. Once the state has been determined, IPC is initialized and the appropriate EHSA call is made.

The sample code below is from `platform_interface_init()` on the Cougar platform. Cougar determines Primary/Secondary roles in the ROM Monitor (rommon). It also has an interrupt that occurs on card status changes:

```
...
asw_ipc_init (aux_line); /* Init IPC, Primary/Secondary role is known */
...
if(this_acpm_state == STATE_MASTER) {
    printf("*** this cpu is the primary\n");
    declare_cpu_good(this_acpm_state);
    issue_system_reset();
    ehsa_event(EHSA_SW_P);
    reg_add_ehsa_switch_to_secondary(cougar_state_change, "cougar state change");
    slo_test(1);
    ehsa_init_info();
    ehsa_primary();
} else if (this_acpm_state == STATE_SLAVE) {
    printf("*** this cpu is the secondary\n");
    declare_cpu_good(this_acpm_state);
    ehsa_event(EHSA_SW_SD);
    reg_add_ehsa_switch_to_primary(cougar_state_change, "cougar state change");
    slo_test(2);
    asw_initiate_fsm();
    ehsa_init_info();
    ehsa_secondary();
    /*
     * if we return here we are becomming the master
     * verify this and continue
     */
    if (ehsa_get_state() == EHSA_PRIMARY) {
        slo_test(1);
        level = get_interrupt_level();
        set_interrupt_level(ALL_ENABLE);
        asw_reinitialize_as_primary();
        reset_interrupt_level(level);
    } else {
        ttyprintf(CONTTY, "ehsa_secondary() returned prematurely\n");
    }
}
```

2.0.1.1 IPC Setup

The initialization of IPC is dependent on state. If a Primary or Standalone, IPC is initialized as master. If a Secondary, IPC is initialized as a slave. This must be done by the platforms.

2.0.1.2 Determining Primary/Secondary Status

The emulation code uses the lowest MAC address to determine Primary/Secondary. However, most platforms should use slot number or config.

2.0.1.3 Platform Initialization of EHSA Information and Vectors

EHSA requires certain information from the platform (the first fields of the structure). The EHSA control structure has the following format:

```
typedef struct ehsa_control_t_ {
    ehsa_platform_oob_send_t   ehsa_oob_send;
    ehsa_poll_crash_ack_t      ehsa_poll_crash_ack;
    ehsa_platform_crash_t      ehsa_platform_crash;
    boolean                    secondary_console_enable;
    boolean                    keepalive_enable;
    ushort                     keepalive_interval;
    ushort                     keepalive_retry_count;
    boolean                    keepalive_failover;
} ehsa_control_t;
```

This structure must be initialized with `ehsa_init_control()` before `ehsa_primary()` or `ehsa_secondary()` is called.

The emulation code does the following before returning from discovery:

```
ehsa_control_t ehsa_control;

    ehsa_control.ehsa_oob_send = ehsa_send_oob_packet;
    ehsa_control.ehsa_poll_crash_ack = ehsa_rx_crash_ack;
    ehsa_control.ehsa_platform_crash = ehsa_platform_crash_handle;
    ehsa_control.secondary_console_enable = TRUE;
    ehsa_control.keepalive_enable = TRUE;
    ehsa_control.keepalive_interval = 10;
    ehsa_control.keepalive_retry_count = 3;
    ehsa_control.keepalive_failover = TRUE;
    if (!ehsa_init_control(&ehsa_control))
        ttyprintf(CONTTY, “\nehsa_init_info(): Error setting ehsa_control”);
```

2.0.2 The Secondary Background Process

This process is created by the `ehsa_secondary()` call to handle EHSA background tasks for the Secondary These include messages (keepalives, etc.) and state changes. This process only lives as long as the state is `EHSA_SECONDAR` .

2.0.3 The Primary Background Process

This process is created by the `ehsa_primary()` call to handle EHSA background tasks for the Primary. These include messages (keepalives, etc.) and state changes. It is killed when the state is no longer `EHSA_PRIMAR` .

2.0.4 Changes in the Initialization Sequence

The initialization sequence has been changed to support EHSA. A new `subsys` class has been added to support subsystems that need to be initialized on the Secondary before the init process is suspended. Process creation is moved up so that the subsystems may run on the Secondary

```
*** old_init.cFri Apr 17 13:06:30 1998
--- new_init.cFri Apr 17 13:05:02 1998
*************** boolean system_init (boolean loading)
*** 73,82 ****
--- 73,112 ----
       * Initialize default generic network support and services
       */
      network_init();

      /*
+      * Initialize the EHSA(redundancy) subsystems.
+      */
+     subsys_init_class(SUBSYS_CLASS_EHSA);
+
+     result = process_create(critical_background, “Critical Bkgnd”,
+     NORMAL_STACK, PRIO_CRITICAL);
+     if (result == NO_PROCESS)
+ return(FALSE);
+     result = process_create(net_background, “Net Background”,
+     NORMAL_STACK, PRIO_NORMAL);
+     if (result == NO_PROCESS)
+ return(FALSE);
+     if (!loading)
+ logger_start();
+
+     /*
+      * Spawn the necessary one second background processes
+      */
+     result = process_create(tty_background, “TTY Background”, NORMAL_STACK,
+     PRIO_NORMAL);
+     if (result == NO_PROCESS)
+ return(FALSE);
+     result = process_create(net_onesecond, “Per-Second Jobs”, NORMAL_STACK,
+     PRIO_NORMAL);
+     if (result == NO_PROCESS)
+ return(FALSE);
+
+     platform_line_init(); /* init serial lines, AUX line and VTYs */
+
+     /*
       * Allow platforms to register special services and functions
       */
      platform_interface_init();

      /*
*************** boolean system_init (boolean loading)
*** 111,121 ****
      /*
       * Activate the subsystem management interfaces
       */
      subsys_init_class(SUBSYS_CLASS_MANAGEMENT);

-     platform_line_init(); /* init serial lines, AUX line and VTYs */

      /*
       * Finish initializations that depend on loaded protocols
       */
      service_init();/* init service booleans */
```

```
--- 141,150 ----
*************** boolean system_init (boolean loading)
*** 128,173 ****
       * Start the major system processes
       */

      set_interrupt_level(ALL_ENABLE);

-     result = process_create(critical_background, “Critical Bkgnd”,
-     NORMAL_STACK, PRIO_CRITICAL);
-     if (result == NO_PROCESS)
- return(FALSE);
-     result = process_create(net_background, “Net Background”,
-     NORMAL_STACK, PRIO_NORMAL);
-     if (result == NO_PROCESS)
- return(FALSE);
-     if (!loading)
- logger_start();
      if (loading) {
  result = process_create(bootload, “Boot Load”, LARGE_STACK,
  PRIO_NORMAL);
  if (result != NO_PROCESS) {
      process_set_arg_num(result, loading);
      process_set_ttynum(result, startup_ttynum);
  }
      }
      reg_invoke_emergency_message(EMERGENCY_SYS_STARTUP);

-     /*
-      * Spawn the necessary one second background processes
-      */
-     result = process_create(tty_background, “TTY Background”, NORMAL_STACK,
-     PRIO_NORMAL);
-     if (result == NO_PROCESS)
- return(FALSE);
-     result = process_create(net_onesecond, “Per-Second Jobs”, NORMAL_STACK,
-     PRIO_NORMAL);
-     if (result == NO_PROCESS)
- return(FALSE);
-     result = process_create(net_periodic, “Net Periodic”, NORMAL_STACK,
-     PRIO_NORMAL);
-     if (result == NO_PROCESS)
- return(FALSE);
-
      return(TRUE);
  }
```

2.1 Common EHSA CLI

This section gives the cross-platform EHSA command line interface (CLI) syntax, which was agreed upon in a cross-BU EHSA working group. The syntax is actually a general framework. Many of the platforms that support EHSA will extend it with platform-specific specializations (new keywords and/or parameters).

The cross-BU EHSA working group has agreed that each EHSA platform will implement its CLI as platform-dependent code and will conform to the common syntax everywhere that it is applicable. There is no platform-independent EHSA CLI implementation at this time.

In addition, the working group has agreed that each platform will update ENG-23371 with its platform-specific syntax extensions. This will encourage platforms adding similar platform-specific features to adopt a common pre-existing syntax. It will especially assist new platforms that are just entering their EHSA d velopment.

2.1.1 Platforms Currently Represented

- Santa (6400) Redundancy CLI (6400)

2.1.2 General Redundancy (EHSA) CLI Syntax

2.1.2.1 Redundancy Configuration

Redundancy configuration is a new configuration submode, containing additional submodes similar to "dialer-profile" or "vp-tunnel" submodes.

```
# redundancy
        #       [no] associate <object-type> <instance#1> [<instance#2>]
        #               [no] <keyword> [<parameter> [<parameter>]...]
        #       main-cpu
        #               [no] <keyword> [<parameter> [<parameter>]...]
        #       switch-fabric <instance#1> [<instance#2> [<instance#3>]]
        #               [no] <keyword> [<parameter> [<parameter>]...]
```

`<object-type>` is a keyword appropriate to the naming of the redundant components on the specific platform, for example, slot, card, and others.

```
<instance> is an specific object identifier, whose syntax depends
on the <object-type> and platform.

<keyword> describes a property of the redundant association, with
optional parameters; the valid set of <keyword>s depends on the
<object-type>.
```

2.1.2.2 Redundancy Displ y

```
> show redundancy [<keyword> [<parameter(s)>]]
```

By default (no keyword), display the redundancy configuration, annotated with the current runtime redundancy status. For example, display which slots are configured to be in redundant mode, which of those slots has a card present, and which of those cards is Primary. Details are platform-specific.

`<keyword> [<parameter(s)>]` may be used to qualify the output requested or to extend the functionality of this command.

2.1.2.3 Redundancy Operations

```
# redundancy <keyword> [<parameter> [<parameter> ...]]
```

Supports runtime operations necessary or desirable to control the behavior of redundant system components, such as forcing failover from the current Primary to the current Secondary element in a redundant association.

`<keyword>` is required to specify the action requested.

2.1.3 Santa (6400) Redundancy CLI

See ENG-23369 for a detailed explanation of the Santa CLI.

```
santa(config)# redundancy
santa(config-r)#        [no] associate slot #1 [#2]
santa(config-r-sl)#             [no] prefer #
santa(config-r)#        [no] associate subslot #1/#1 [#2/#2]
santa(config-r-su)#             [no] prefer #/#
santa(config-r)#        [no] associate port #1/#1/#1 [#2/#2/#2]
santa(config-r-p)#              [no] prefer #/#/#
santa(config-r-p)#              [no] aps protection #1/#1/#1
santa(config-r)#        main-cpu
santa(config-r-mc)#             prefer <A|B>
santa(config-r-mc)#             [no] sync config <startup|running|both >


santa> show redundancy

santa# redundancy force-failover <slot # | subslot #/# | port #/#/# | main-cpu  >
```

2.2 EHSA Crash Handling

This section reviews all additions that need to be made to the platform code and the independent code in order to support EHSA software crash handling.

Note The EHSA crash handling support is available only on MIPS-based platforms.

2.2.1 Background

There are several possible scenarios where a software crash could occur, including the following:

1. `crashdump()`—This function is called explicitly by the code when the software has gotten into an invalid state and chooses to crash the box.
2. exception—In case of a software exception, the MIPS code checks whether the corresponding exception entry was initialized with a call-back function by the platform code. If the entry was initialized, then the platform exception function is called; otherwise, the function `handle_exception()` is called.

   In `handle_exception()`, the stack trace information is sent, a core is dumped if the box is configured to do so, and the software returns to the Rommon.

Some software hooks have been added to these functions that are currently executed during a box crash, `crashdump()` and `handle_exception()`, to support EHSA crash handling. These hooks are described below.

Note If the platform provides another possible crash path other than the two mentioned, then the same hooks will have to be added to those routines. It is up to the platform to add that code.

2.2.2 What happens when a Primary crashes?

This section describes the actions that take place in the system when the software on the Primary crashes. It describes functional behavior, in chronological order, on both the Primary and Secondary processors.

1 The Primary identifies a software crash, either in `crashdump()` or in an exception routine. The `ehsa_crash_hook()` routine is invoked via registry. This routine sends a message to the Secondary to indicate that the Primary is crashing and that the Secondary should take over. The message is sent by an out-of-band (OOB) transmit routine. At that time, it also sets a timer that defines the maximum time interval that it will wait before it will continue with the crash. The timer is currently set to ONESEC. It can be converted to a variable if required.

Note It is up to the platform to provide an OOB transmit routine with which to send the message to the Secondary. The message can be sent on any media that the platform chooses except for IPC. Because the interrupts are disabled when a crash occurs, IPC is not reliable at that time.

The EHSA code sends the crash message once. However, a hook has been provided for an platform that requires a more reliable mechanism and chooses to resend the message. It is up to the platform to add the code for retransmission.

2 The Secondary gets the message from the Primary that indicates that the Primary is crashing. The Secondary becomes the new Primary.

In order to become the new Primary, it needs to perform a Secondary-to-Primary switchover. Please refer to the Implementation Guide earlier in this chapter for more details about the switchover.

Then it will send an EHSA message to the old Primary indicating that the new Primary has taken over.

3 The old Primary waits until it either receives the ack message or a timer expires. The platform should provide a polled receive routine that receives that message.

The old Primary calls the `ehsa_platform_crash()` routine. That routine is provided by the platform, and will be called via a function vector. The function sends crash information to wherever the platform decides to send it. That is the right place for additional functionality required by the platform before the old Primary returns to Rommon.

The old Primary boots the IOS image and becomes the new Secondary. It is up to the platform to insure that the processor becomes Secondary and boots as a Secondary.

It is possible that a problem in the code or the configuration will not enable the IOS image to boot. If that happens, we might run into a situation where we have constant swaps between Primaries. That can start an infinite swap chain, since the IOS image can never fully boot. It is suggested that the platform provide a mechanism to prevent this possibility.

The platform can hold a counter to count the number of boot trials in a given amount of time. If the number of boot trials exceeds a predefined constant, the platform should indicate so. That counter can be held in the NVRAM, in the area just before the configuration. It is up to the platform to decide how to handle this case. It can try to boot from another device or it can stop the attempts to boot.

2.2.3 What happens when a Secondary crashes?

In the case of a Secondary crash, the following is done:

1. The Secondary sends a message to the Primary to notify it that a crash has happened. That is done using the same OOB transmit routine used in the case of a Primary crash.
2. Crash information is sent by the Secondary. It is up to the platform to decide what type of information should be sent and where.
3. The Secondary returns to Rommon and reboots as a Primary.

2.2.4 Summary of Routines and Code Additions

These are the routines and code additions required for implementing EHSA crash handling:

1. A registry called `ehsa_crash_hook()` has been added. The `reg_add_ehsa_crash_hook()` call has been added to the EHSA initialization section. This call registers the `ehsa_crash_hook()` to be called in case of a master crash.

   This registry will be invoked in any of the following cases:

   (a) In the beginning of a `crashdump()` routine. The `reg_invoke_ehsa_crash()` will be followed by a call to `crashpoint()`, which will go back to Rommon via an `emt()` call.

   (b) In the `handle_exception()` routine, just after it recognizes that this is not a GDB exception. This call will be followed by a call to `r4k_return_to_monitor()`.

   (c) If any of the platform exception handlers is crashing the software by going back to Rommon, the registry `reg_invoke_ehsa_crash()` should be added in the appropriate place, just before the function returns to Rommon.

   The `reg_invoke_ehsa_crash()` call has been added to the platform-independent code, that is items a and b in the above list. It is the platform's responsibility to add this registry to any appropriate exception routine, as explained above in item c.

   The `ehsa_crash()` call will identify that it is a Primary crash and will send a message to the Primary using an OOB transmit routine. It will then wait till it receives a message from the new Primary indicating that the Primary switchover has been done.

2. Two new vector functions have been added to the EHSA control structure. The platform code needs to provide these functions. The platform code should initialize the appropriate field in the control structure as part of the EHSA platform initialization.

   The functions to be provided by the platform are as follows:

   – `* ehsa_poll_crash_ack()`

     This is a polled receive routine that receives the message that is sent by the other side, to acknowledge the receipt of the crash message.

   – `* ehsa_platform_crash()`

     The function sends crash information to wherever the platform decides to send it. The crash information includes a stack trace and a core file. Usually during a crash we also try to flash the logging buffer. It is up to the platform to decide how and where to send the crash information. It could decide to store the stack trace locally on the dying Primary flash memory. Or it could choose to try to send it to the new Primary, or somewhere else. The same goes for the core file; the platform could choose to send the core file information to the new Primary. Since it is up to the platform to decide where to send the crash information, the platform will have to provide the code that sends it.

PART 2

Kernel Services

CHAPTER 3

Scheduler

The Cisco IOS scheduler is responsible for scheduling kernel processes for execution. The scheduler is nonpreemptive, so all processes must voluntarily relinquish control to the scheduler. In general, a process should perform a small amount of work, relinquish the processor, and then continue working the next time it receives control from the scheduler. There are several constants in the system that determine what is appropriate behavior for processor use.

The scheduler supports multiple process priorities.

The Cisco IOS scheduler uses the list manager to keep track of all processes in the system. The scheduler maintains one list per process priority or process type. This allows it to easily determine which process should be scheduled next. The kernel identifies each process by its process number, called a process identifier (process ID, or PID).

After giving an overview of scheduler-related changes in Release 12.0 that enhance software scalability, this chapter describes how the scheduler works.

3.1 Scalability Changes

In pre-12.0 Cisco IOS code, polling mechanisms would search through all IDBs on a frequent basis, looking for work that needed to be done. Because the number of IDBs rose steadily at a steep rate, this method of scheduling became too costly. In Release 12.0, changes were made to mitigate these costs. They are described in this section:

- Important Coding Guidelines
- if_onesec Registry Removed
- Event Driven Route Adjustment Message
- API for Keepalive and Other Periodic Intervals
- FYI: Backup System Changes

3.1.1 Important Coding Guidelines

To maximize the scalability of your code, be sure to follow these guidelines:

- Do not write code that needs to do loops on every interface, every second. Do write code that uses a subblock list or private IDB list instead (a list of interfaces that you are interested in).
- Do not write code that polls. Do write code that is event driven. This includes using managed timers instead of polling a passive timer.
- Onesec routines (poll every second) are okay, as long as they are short.

3.1.2 if_onesec Registry Removed

The following statements summarize the changes that were made in Release 12.0 to make several one-second polling implementations more scalable:

- atm_arp_onesec()
  This routine has been converted into a watched boolean and moved into an ATM-specific, subsystem-created process. [sent email to Mike Davison 6/26]
- atm_arp_serv_periodic()
  This routine already consisted of managed timers. It is now a onesec routine that loops through the list ofATM interfaces.
- cbus_restart_check()
  This is a 7000-platform-specific routine. The 7000 platform was removed in Release 12.0.
- channel_onesec()
  This CIP routine is now a onesec routine that loops through the subblock list.
- backup_timers and backup loads
  FYI: Backup system feature code was reworked to enhance scalability as well as maintainability and performance. For an overview of the system's design and use and a description of all items in the rework, see ENG-14273.
- Three IP routing routines:
  - ip_gdp_periodic()
    This routine has been changed into either managed timers or a private SWIDB list. Include it only when IP and GDP are both turned on.
  - ip_irdp_periodic()
    This routine has been changed into either managed timers or a private SWIDB list. Include it only when IP and IRDP or mobile beaconing are both enabled.
  - standby_check_if_reset()
    This routine has a flag that indicates when an interface reset has been requested. It has been moved into a higher, common routine.
- Two WAN routines (two instances with the same function name) are now onesec routines:
  - quicc_serial_onesec_periodic()
    This routine was called to update LEDs. It has been moved to a common periodic routine that is subsystem-invoked.
- Seven routines in the LS1010 code were changed into an LS1010-specific if_onesec routine.
  - mmc_poll_invalid_cells()
    Was changed to a private/subblock IDB list and put into a common LS1010 routine. (Only operates on ATMS2000 IDBs.).
  - dcu_onesec()
    Performs keep-alives. Moved to an LS1010 common routine.
  - oc3suni_rd_cntrs(), sali25_ISR(), suni622_rd_cntrs(), sunipdh_rd_cntrs()
    All read counters and update the hardware IDB statistics. Were moved to a common LS1010 routine.
  - In the LS1010/Rhino code, a call to usecdelay()
    This routine essentially just spins its wheels in a loop watching an 8254 timer. Solution unknown at this point. It is possible that this routine is implemented only on a single control card, in which case it is okay.

- One ATM card routine, c2katm_if_onesec()
  This routine is now a onesec routine.
- ppp_timer()
  5 of 6 timers used here have been converted into managed timers and serviced by a separate process; is now event driven, using a managed timer.
- serial_pulsetimer()
  Timer has been moved into the serial subblock; converted to a managed timer.
- serial_restart()
  Converted to a managed timer.

3.1.3 Event Driven Route Adjustment Message

The `route_adjust()` call has been removed from the `net_periodic()` routine. The `net_periodic()` routine is a once-per-second process that is basically a wrapper for the `periodic_activity()` function, a polling routine that, among other things, is responsible for the following two things:

- Checking dialer links.
- Calling `transition_adjust()` and then `route_adjust()` and, if a transition has occurred, a few other routines.

Because dialer links never go down, the normal interface notification routines had no effect. The major scalability problem, though, was with the `route_adjust()` call, which cycled every software IDB in the box and enqueued a `route_adjust_msg()` in a watched queue. All processes watching that queue would have work to do when they awoke. Because the amount of work that was needed was already being taken care of by other routines, the `route_adjust()` call was a waste of time except for the following routines:

```
ipaddress.c
lec_parser.c
atm_dxi.c
dialer.c
tring.c
frame_relay.c
```

To address the scalability problem, the `route_adjust()` call has been moved from the `periodic_activity()` function to inside the `if_(transition_occurred)` conditional and `route_adjust_msg()` calls were moved into the six routines listed above. In this way, the need for `route_adjust()` to be called every second was removed, thereby transforming `route_adjust()` from a polled to an event-driven mechanism.

Note Because the polling mechanism will no longer generate `route_adjust()` calls every second, it is your responsibility to generate your own as necessary by adding a `route_adjust_msg()` call to the appropriate place in your code.

3.1.4 API for Keepalive and Other Periodic Intervals

In pre-12.0 releases of the Cisco IOS software, the `periodic_activity()` subroutine is called for each hardware IDB every second. Among other things, the `periodic_activity()` routine decrements the keepalive counter, `idb->keep_count`, and, if the keepalive interval has expired

(`idb->keep_count <= 0`), it resets the keepalive counter using `idb->keep_period` and calls two per-IDB vectors: `idb->periodic` and `idb->device_periodic`. Therefore, the per-IDB periodic and device periodic functions are called every `keep_period` number of seconds.

This is a classic polled implementation. It searches through all IDBs looking for an expired time that indicates work to do. In our environment of fast increasing number of interfaces, this type of implementation does not scale well.

Further, the pre-12.0 implementation contains inherent opportunities for wasteful cycles:

- If `idb->keep_period` is zero, both periodic vectors are called every second, not every keepalive period. While there is a field to disable keepalive in the IDB, some code sets the keepalive interval to zero while leaving keepalive enabled.
- Some drivers or subsystems set the device periodic and periodic vectors to the same function. This means that the periodic vector is called twice each keepalive period.
- Some drivers or subsystems set both periodic vectors to NULL. Although the calling code checks for this, that check is required to determine that there is nothing to do. (Another NULL function, `return_nothing()` is sometimes used.) Almost no drivers or subsystems use both vectors.

3.1.4.1 New Implementation

To improve the scalability of periodic processing, polling on `idb->keep_count` has been converted to a managed timer implementation. A tree of keepalive interval managed timers has been created that has a parent timer for each unique keepalive period.

These keepalive timers are parented to the `net_background_time` , so the first requester to set a ne unique keepalive interval creates a new keepalive parent timer. Since the number of unique keepalive intervals tends to be small, this implementation scales well. The creation of these parent timers allows the system to restart a given timer with minimal overhead as it is reinserted into the scheduler's managed timer chain.

Eight new API functions have been created for getting information about the period (interval set and time left before the next vector will be called), enabling or disabling keepalive frames, and determining whether keepalive frames are enabled:

Table 3-1 New API Functions for Keepalive Frames and Other Periodic Intervals

`idb_hw_set_periodic()`	Set the periodic vector
`idb_hw_set_device_periodic()`	Set the device periodic vector
`idb_set_periodic_period()`	Set the periodic interval
`idb_get_periodic_period()`	Determine the length of the periodic interval
`idb_get_periodic_period_left()`	Determine the time left before the next vector will be called
`idb_enable_keepalives()`	Enable keepalive frames
`idb_disable_keepalives()`	Disable keepalive frames
`idb_are_keepalives_enabled()`	Determine whether keepalive frames are enabled

Use these API functions instead of changing the vectors directly. This conversion is not backward compatible; the names of the pertinent IDB vector fields have changed. For example, the `idb->periodic` field is now `idb->periodic_fn` and the `idb->device_periodic` field is now `idb->device_periodic_fn`. (Table 3-2 lists the name changes.) Code using the old method will not compile until it has been converted to use the new keepalive/periodic API functions.

3.1.4.2 Setting the Periodic Interval

Use the following function to set the periodic interval:

```
idb_set_periodic_period (hwidbtype *hwidb, ulong period)
```

Note that the periodic period is now maintained in ticks, not seconds. Use the ONESEC macro, passing in the number of ticks corresponding to *X* seconds:

```
(X * ONESEC)
```

For portability reasons, do not use absolute "magic" numbers of ticks that assume a certain tick interval.

Use the following function to set the `idb->periodic_fn` vector:

```
void idb_hw_set_periodic (hwidbtype *hwidb, periodic_t periodic_function)
```

Pass the function that you wish to be called to the hardware IDB that represents your interface.

The `periodic_fn` vector performs various periodic functions needed by an interface. Currently it is called once every periodic interval (the default is every 10 seconds). Normally it is used by level-2 encapsulations, such as PPP. It is not used by most device drivers.

Warning The `idb->periodic_fn` vector may be modified by encapsulation routines.

Use the following function to set the `idb->device_periodic_fn` vector:

```
void idb_hw_set_device_periodic (hwidbtype *hwidb,
                                 device_periodic_t device_periodic_function)
```

Pass the keepalive or other periodic functions that you wish to be called to the hardware IDB that represents your interface.

The `device_periodic_fn` vector performs various device-dependent background tasks, such as making sure that the physical device is still alive. Only the device driver is allowed to modify this vector.

If both vectors for a given IDB (periodic and device periodic) are NULL, the 11.3 code notices it and does not run a timer for that IDB. If either or both vectors are set at a later time, the code will also notice and start the timer for that IDB.

Use the following function to determine the periodic interval on which the `periodic_fn` and `device_periodic_fn` vector functions are called:

```
static inline ulong idb_get_periodic_period (hwidbtype *hwidb)
```

Note that the periodic interval will be returned in ticks, not seconds, and that the name has been changed from "keepalive interval" to "periodic period."

Use the following function to determine the time remaining until the periodic functions will be called:

```
ulong idb_get_periodic_period_left (hwidbtype *hwidb)
```

The time remaining will be returned in ticks, not seconds.

3.1.4.3 Setting Keepalive Frames

Use the following functions to enable or disable keepalive frames:

```
void idb_enable_keepalives (hwidbtype *hwidb)
void idb_disable_keepalives (hwidbtype *hwidb)
```

Use the following function to determine if keepalive frames are enabled for your interface:

```
static inline boolean idb_are_keepalives_enabled (hwidbtype *hwidb)
```

The returns TRUE if keepalives are enabled. Note the polarity change from the former `nokeepalive` IDB flag (which was TRUE if keepalives were disabled).

3.1.4.4 Hardware IDB Field Name Changes

Driver conversion to the new keepalive/periodic API is straightforward unless a driver was accessing the `hwidb->keep_count` field. That field is no longer directly accessible.

Table 3-2 Hardware IDB Field-Name Changes

Pre-11.3 Field	11.3 Field
`short keep_period`	`ulong periodic_interval`
`short keep_count`	`mgd_timer periodic_timer`
`tinybool nokeepalive`	`tinybool keepalive_flag`
`periodic_t periodic`	`periodic_t periodic_fn`
`device_periodic_t device_periodic`	`device_periodic_t device_periodic_fn`

3.1.5 FYI: Backup System Changes

Passive timers in the backup code were replaced by managed timers and an existing background service routine was modified to service the new (managed) backup timers. Backup code was rewritten for Release 12.0. See Appendi x27 for an overview of the system and full description of the rewrite.

3.2 Processes: Overview

A *process* in Cisco IOS terminology is roughly the equivalent of a *thread* in computer science terminology. A Cisco IOS process consists of a set of processor registers and a stack area. All processes share the same text segment, executing from mostly nonoverlapping code paths. They also share the same flat memory space, and thus share all variables. Cisco routers do not currently support any form of virtual memory or memory mapping. (The R4000-based processors have user and kernel data segments, but these are a complete overlay, and all memory is accessible from either processor mode.)

3.2.1 How a Process Is Created

New processes can be created at any time by any process. Processes are generally created either as part of the startup process—from either the `init` process or a subsystem initialization routine—or they are created in response to a user configuration command. The process creation function assumes that the process takes no argument and does not have a terminal attached to it. If the process has an argument or a terminal, it is informed of it by a call to one of the process attribute modification functions.

When a process is created, it is assigned a positive PID number. A PID of 0 is never used.

3.2.2 How a Process Stops

A process stops executing by killing itself. The `main` routine of a process must explicitly call the `process_kill()` function; it cannot just execute a `return` statement. The latter is considered an error condition and is protected against. When processes are no longer needed—for example, when a protocol is unconfigured—they should clean up after themselves and exit.

3.2.3 How the Scheduler Executes a Process

The scheduler executes a process by calling a special function that saves the scheduler's current state, restores the state of the process from a data structure, and then returns. This return occurs on the process' stack and therefore causes the process to continue executing. When the process relinquishes the processor, it calls a second special function that saves the process state, restores the scheduler state, and then returns. The return from this function happens on the scheduler's stack and thus causes the scheduler to continue executing where it was before it ran the process.

3.2.4 Process States

All processes exist in one of a finite set of states that describe their current activit y. Table3-3 describes these states.

Table 3-3 Process States

Process State	Description
Running	Process is currently executing in the CPU.
Suspended	Process has temporarily relinquished the processor. It will resume at the next possible opportunity. Processes enter this state to allow other processes to use the processor. Because the Cisco IOS scheduler is a run-to-completion scheduler, if a process did not suspend occasionally, nothing else would run.
Ready to run	Conditions for executing have been fulfilled, and the process is in the ready queue, waiting to run. This implies that the process was previously waiting for an event or a timer, and that that event or timer has occurred.
Waiting for event	Process is awaiting completion of an event before being ready to execute.
Sleeping (absolute time)	Process is suspended until a specific clock time.
Sleeping (interval)	Process is suspended until a specific time interval has elapsed.

Process State	Description
Sleeping (periodic)	Process is suspended until a regular time period has elapsed. This state is essentially the same as "sleeping (interval)," with one subtle difference. With "sleeping (periodic)," the wakeup time is computed based on the time the process began executing, not the time the process finished executing. This allows a process to execute at a fixed interval and not have its wakeup time slowly drift because of processing time. For example, suppose a process should execute every second, and it requires 0.1 seconds to perform its processing. Using the `process_sleep_periodic()` function, the processes would execute at times 0.0, 1.0, 2.0, 3.0, and so on. Using the `process_sleep_for()` function, which places a process in the "sleeping (interval)" state, this same process would execute at times 0.0, 1.1, 2.2, 3.3, and so on.
Sleeping (managed timer)	Process is suspended until a managed timer has expired.
Hung	Process would not relinquish the processor and was stopped by the watchdog interrupt. This process will never again be scheduled to receive the processor.
Dead	Process has stopped or been killed and will never resume. This is a transient state that lasts until the scheduler can perform a postmortem analysis on the process and reclaim its resources.

3.2.5 Scheduler Messages

The scheduler uses messages to provide a simple interprocess communication (IPC) mechanism that works on a single processor. A *message* is a simple way for two processes to communicate. message consists of a command and two arguments, a numeric argument and a pointer argument. Messages are used with registry processing. For example, you might use a message to indicate that an interface has changed state. For this example, the command code would indicate "interface state change," and either the numeric or pointer argument would indicate the interface.

The scheduler messages are in addition to the standard Cisco IOS IPC services, which are discussed in the "Interprocess Communications (IPC) Services" chapter.

3.3 Queues and Process Priorities

Process priority controls how often a process gets processor (CPU) time. The scheduler uses multiple queues to track the processes that run various priorities.

3.3.1 Scheduler Queues

The scheduler provides three types of queues:

- Ready queues
- Idle queue
- Dead queue

The scheduler queues are implemented using the Cisco IOS list manager, which is described in the "Queues and Lists" chapter.

3.3.1.1 Ready Queues

Ready queues are used for processes that are ready and waiting to run. A process is in a ready queue if at least one of the process' conditions for executing, if there were any, has been fulfilled.

Ready queues can handle processes of critical, high, medium, and low priority.

3.3.1.2 Idle Queue

The idle queue is for processes that are waiting for an event to occur before they can execute. These processes are actively awakened by the code fragment that creates them. As an example, the code fragment that enqueues a packet on the VINES receive queue is responsible for waking up the VINES process. The VINES process does not poll the queue at every pass of the scheduler.

The event that causes the process to execute can be the expiration of a timer, or it can be an asynchronous event, such as a packet being enqueued into a specific queue.

All processes in the idle queue that are awaiting a timer event are threaded by expiration time into a tree maintained by the managed timers code. (Managed timers are discussed in the "Timer Services" chapter.)

3.3.1.3 Dead Queue

The dead queue is for processes that have exited, but on which the scheduler has not yet performed a postmortem analysis. When the scheduler processes this queue, it analyzes the stack for each process and then releases all memory associated with its record of that process.

3.3.1.4 Moving Processes between Queues

During its operation, the scheduler moves processes between two queues, for example, from the idle queue to the ready queue. Consider the following points concerning moving processes between queues:

- All linking and unlinking is executed with interrupts disabled.
- A running process is unlinked from the queue only when it executes a scheduler primitive to relinquish the processor.
- A running process is not moved to the idle queue (either from a sleep or wait call) if new events have arrived for it while it has been executing. Instead, the process is left on the ready queue so that it can process the new data in the next pass of that queue. This protects against the race condition that can occur if a process is relinquishing the processor at the same time that an interrupt-level code path enqueues data for the process.

3.3.2 Process Priorities

Processes are grouped according to their priority. Processes can run at one of the following priority levels:

- Critical. This priority is for processes that resolve resource allocation problems, for example, a process that creates or replenishes the packet buffer pools.

Caution A single, poorly behaved process running at critical priority can disable a router, because an item on this queue has the chance to execute after every high-, medium-, and low-priority process. Therefore, the use of this priority level *must* be approved by an engineer in the Cisco IOS Technology group.

- High. This priority is for processes that need more than an average amount of CPU time, such as processes that accept packets directly from a network interface.
- Medium. This is the default priority in which most processes, such as EXEC commands and routing protocols, should execute.

- Low. This priority is generally used for periodic background tasks, such as logging and the TCP discard daemon.

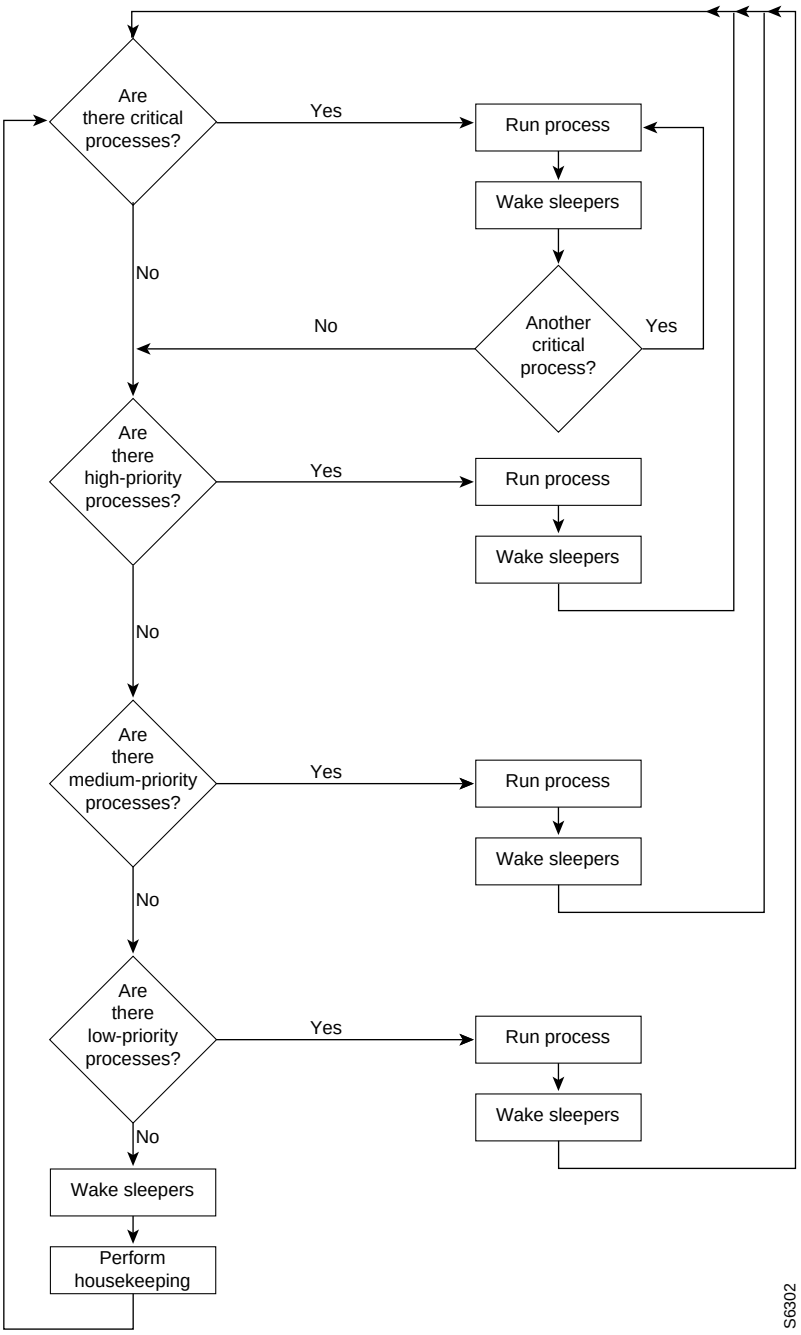
3.3.3 Operation of Scheduler Queues

In the Cisco IOS scheduler, the queues are not given equal processing time. Instead, the high-priority and critical-priority queues are processed multiple times for each pass at the medium-priority and low-priority queues.

The scheduler queue operation is as follows. Figure3- 1 illustrates this operation.

1. Run all processes in the critical-priority list.
2. Run one process in the high-priority list.
3. Repeat Steps 1 and 2 until no critical-priority or high-priority processes remain to be run.
4. Run one process in the medium-priority list.
5. Repeat Steps 3 and 4 until no critical-priority, high-priority, or medium-priority processes remain to be run.
6. Run one process in the low-priority list.
7. Repeat Steps 5 and 6 until no processes remain to be run.
8. Wake sleeping processes. All processes are threaded via managed timers. The scheduler checks the parent timer for expiration time and moves expired processes to the appropriate run queues.
9. Perform housekeeping operations. These include computing CPU loads and busy times, performing postmortem analyses on processes, performing “scheduler-interval” processing, and spinning a random-number generator.

Figure 3-1 Overall Scheduler Queue Operation



3.4 Manage Processes

3.4.1 Create a Process

All processes are created using the `process_create()` function. This function does not pass any arguments to the process and does not provide the process with a controlling terminal.

```
pid_t process_create(process_t (*padd), char *name, stack_size_t stack,
                     process_priority_t priority);
```

Once a process has been created, use either the `process_set_arg_num()` function to provide a numeric argument or the `process_set_arg_ptr()` function to provide a pointer argument.

```
static inline boolean process_set_arg_num(pid_t pid, ulong arg);

static inline boolean process_set_arg_ptr(pid_t pid, void *arg);
```

For processes that require a controlling terminal, such as access server and EXEC processes, use either the `process_set_ttynum()` or `process_set_ttysoc()` function to provide the terminal.

```
static inline boolean process_set_ttynum(pid_t pid, int ttynum);

static inline boolean process_set_ttysoc(pid_t pid, tt_soc *tty);
```

3.4.1.1 Create a Process: Example

The following example creates a process for the TCP discard daemon, which is run from the `tcpdiscard_daemon` routine. The process is named `TCP Discard`. It has a process stack size of `NORMAL_STACK` and runs at low priority. After the process has been created, the `process_set_arg_ptr()` function passes it an argument.

```
tcp->pid = process_create(tcpdiscard_daemon, “TCP Discard”, NORMAL_STACK, PRIO_LOW);
if (tcp->pid != NO_PROCESS) {
    process_set_arg_ptr(tcp->pid, tcb);
} else
    tcp_abort(tcp);
}
```

The following example creates a bootload process, passing an argument to the process and assigning the console as the output device:

```
result = process_create(bootload, “Boot Load”, LARGE_STACK, PRIO_NORMAL);
if (result != NO_PROCESS) {
    process_set_arg_num(result, loading);
    process_set_ttynum(result, startup_ttynum);
}
```

3.4.2 Enqueue Data for a Process

To add an item for a process to the end of a managed queue and awaken any processes watching this queue, use the `process_enqueue()` function.

```
boolean process_enqueue(watched_queuetype *queue, void *whatever);
```

To add a packet or an arbitrary data pointer to the beginning of a managed queue and awaken any processes watching this queue, use the `process_requeue()` function.

```
boolean process_requeue(watched_queuetype *queue, void *whatever);
```

To add a packet or an arbitrary data pointer to the beginning of a managed queue and awaken any processes watching this queue, use the `process_requeue_pak()` function.

```
void process_requeue_pak(watched_queuetype *queue, paktype *pak);
```

3.4.3 Dequeue Data from a Process

To remove an item for a process from the beginning of a managed queue, call the `process_dequeue()` function.

```
void *process_dequeue(queuetype *queue);
```

3.4.4 Register a Process for Notification on a Timer

To register a process to be notified that a timer has expired, use the `process_watch_mgd_timer()` function for managed timers or the `process_watch_timer()` function for simple (passive) timers. (Timers are discussed in the "Timer Services" chapter.)

```
void process_watch_mgd_timer(mgd_timer *timer, ENABLEDISABLE enable);

void process_watch_timer(sys_timestamp *timer, ENABLEDISABLE enable);
```

When the timer expires, the process that is watching it is awakened. A process can watch a single managed timer and a single simple timer at the same time. It cannot watch multiple simple timers simultaneously. Even though a process can watch only one managed timer directly, an arbitrary number of managed timers might be subordinate to that single watched managed timer.

For simple timers, the wakeup time is read when the process relinquishes the CPU and is not dynamically updated. In contrast, the wakeup time of a managed timer can be changed while a process is sleeping, and the new wakeup time automatically propagates into the scheduler.

3.4.5 Set and Retrieve Information about a Process

Table 3-4 and Table 3 -2 list the functions available to set and retrieve information about processes.

Table 3-4 Set and Retrieve Information about a Process

Information about Process	Function to Set	Function to Retrieve
Name	`process_create()` `process_set_name()`	`process_get_name()`
Identifier (PID)	—	`process_get_pid()`
Priority	`process_create()`	`process_get_priority()`
Controlling terminal	`process_set_ttynum()` `process_set_ttysoc()`	`process_get_ttynum()` `process_get_ttysoc()`
Profiles	`process_set_profile()` `process_set_all_profiles()`	`process_get_profile()`
Stack size	—	`process_get_stacksize()`
Classes of events allowed to wake up a process	`process_set_wakeup_reasons()`	`process_get_wakeup_reasons()`
Next reason why a process was awakened	—	`process_get_wakeup()`
Starting time	—	`process_get_starttime()`
Cumulative running time	—	`process_get_runtime()`

Information about Process	Function to Set	Function to Retrieve
Arguments that are passed to process	`process_set_arg_num()` `process_set_arg_ptr()`	`process_get_arg_num()` `process_get_arg_ptr()`
Whether to perform postmortem analysis of process	`process_set_analyze()`	`process_get_analyze()`
Whether process should stop running while a core dump is being written	`process_set_crashblock()`	`process_get_crashblock()`

3.4.6 Send a Message to a Process

To send a generic message to a process, use the `process_send_message()` function.

```
boolean process_send_message(pid_t pid, ulong id, void *ptr_arg, ulong num_arg);
```

3.4.7 Retrieve Messages for a Process

To retrieve the next message that is waiting for this process, use the `process_get_message()` function.

```
boolean process_get_message(ulong *id, void **ptr_arg, ulong *num_arg);
```

3.4.8 Determine Whether a Process Exists

Two functions—`process_exists()` and `process_is_ok()`—allow you to determine whether a process exists. Both functions determine whether a process identifier (PID) exists; in addition, `process_is_ok()`, if `TRUE`, indicates that the process has not failed.

```
boolean process_exists(pid_t pid);

boolean process_is_ok(pid_t pid);
```

3.4.9 Suspend a Process

Table 3-5 and Table 3 -2 list the functions available for suspending a process.

Table 3-5 Functions for Suspending Processes

Purpose	Effect	Function
Suspend unconditionally.	Place the process on appropriate ready queue. The process executes again during the scheduler's next pass of that queue.	`process_suspend()`
Conditionally relinquish the processor.	Execute the suspended process again as soon as all other ready processes at the same or higher priority have executed.	`process_may_suspend()`
Check whether a process has exceeded its allotted time (time quantum).	—	`process_time_exceeded()`

Purpose	Effect	Function
Suspend for a finite period of time.	Execute a process again as soon as the specified time has been reached or as soon as the time interval has elapsed. The process will not wake up for any other reason.	
	Sleep for the specified amount of time (number of milliseconds).	`process_sleep_for()`
	Sleep until a managed timer expires. This is the preferred method if the process uses managed timers or if the timers within the process will be modified by any other process. The wakeup time of the modified managed timer percolates from the process' private timers up into the scheduler, thereby automatically adjusting the wakeup time of the process.	`process_sleep_on_timer()`
	Sleep for a specified amount of time (number of milliseconds) from the last time the process should have awakened. This allows processes to begin execution at a fixed time interval, even though a single execution might have lasted longer than usual because the processor was temporarily busy.	`process_sleep_periodic()`
	Sleep until an absolute time is reached.	`process_sleep_until()`
Suspend until an asynchronous event occurs.	Place a process on the idle queue and ignore it until the process is explicitly awakened by another process, an event for which the process has registered occurs, or a timer with which the process has registered has expired.	
	Suspend the process until an asynchronous event occurs.	`process_wait_for_event()`
	Suspend the process, providing a temporary timer to limit how long the process can remain idle.	`process_wait_for_event_timed()`
Determine whether a process can suspend.	Have a process determine whether it is in a context in which it is allowed or desirable to suspend itself.	`process_suspends_allowed()`
	Determine whether a high-priority or critical-priority process is ready to run or whether the current process' quantum has expired.	`process_would_suspend()`

3.4.10 Wake Up a Process

The scheduler provides two functions to wake up a process—`process_wakeup()` and `process_wakeup_w_reason()`. Waking up a process moves it to the appropriate ready queue. Most processes are awakened as a result of the wakeups implicit in modifying a managed data structure. Therefore, you rarely need to call these functions directly.

```
void process_wakeup(pid_t pid);

void process_wakeup_w_reason(pid_t pid, ulong major, ulong minor);
```

To determine the classes of events that are allowed to wake up the current process, use the `process_get_wakeup_reasons()` function.

```
static inline boolean process_get_wakeup_reasons(ulong * reasons);
```

To determine the next reason why the current process has been awakened, use the `process_get_wakeup()` function.

```
boolean process_get_wakeup(ulong *major, ulong *minor);
```

3.4.11 Delay a Process

The scheduler provides two functions to delay a process.

The `process_wait_on_system_init()` function causes the process to block and wait until the system initialization flag becomes `TRUE`. This is a simple function that indicates that the process wants to be informed when a boolean changes state, idles until that event occurs, and then indicates that the process no longer needs notification of the event. Using this function saves code space.

```
void process_wait_on_system_init(void);
```

The `process_wait_on_system_config()` function halts a process until all interfaces in the router have been configured. In all other respects, it operates in the same way as `process_wait_on_system_init()`.

```
void process_wait_on_system_config(void);
```

3.4.11.1 Delay a Process: Example

Use the `process_wait_on_system_init()` function at the beginning of a process that should not run during system initialization. For example, the normal starting code for packet-handling routines is similar to this:

```
void snark_input (void) {
    process_wait_for_system_init();
    reg_add_raw_enqueue(snark_enqueue, “snark_enqueue”);
    while (snark_running) {
        ...
    }
}
```

The code in the first line of this example prevents `snark_input()` from running until the system is initialized. Delaying registration with `raw_enqueue()` means that all packets of type `snark` are sent to `net_input` and discarded until the system finishes initializing. This example code replaces a spin loop where the process discards packets or a call to `systeminitBLOCK`, which is an older method of doing the same thing.

3.4.12 Destroy a Process

To destroy a currently running process, call the `process_kill()` function. Specify as the argument the process identifier of the process to kill or the constant `THIS_PROCESS`, which kills the process that is currently executing.

```
boolean process_kill(pid_t pid);
```

3.5 Scheduler Objects: Overview

The scheduler provides primitives for managing four types of objects: queues, booleans, bit fields, and semaphores. The primitives perform such operations as creating and deleting the objects, and setting processes to be notified when the object is modified.

3.6 Manage Queues

3.6.1 Queue: Definition

A *queue* is a simple data structure for maintaining a linked list of objects. It is an ordered collection of items that keeps track of the first and last objects, the current number of objects, and the maximum number of objects.

3.6.2 Create a Watched Queue

Creating a queue that can be managed by the scheduler allows processes to be awakened automatically whenever new items are enqueued. To create a queue, use the `create_watched_queue()` function.

```
watched_queue *create_watched_queue(char *name, int limit, ulong id);
```

3.6.3 Modify the Queue Minor Type

Use the `process_set_queue_minor()` function after creating a watched queue and before watching it to modify the minor type that the queue receives when the queue state changes. This function is most useful when the queue is created in a common function that specifies a default minor identifier for each queue it creates. The `process_set_queue_minor()` allows a process to unambiguously re-identify queues created in this way before watching them.

```
void process_set_queue_minor(watched_bitfield *event, ulong minor);
```

3.6.4 Register a Process for Notification on a Watched Queue

Some processes, such as those awaiting the receipt of packets, need to be notified when packets arrive on the specified queue. To register the process to be notified that an item has been added to a queue, use the `process_watch_queue()` function. In this function, you specify the frequency of the notification; that is, whether the process should be awakened only the first time packets arrive on the queue (`ONESHOT`) or every time packets arrive (`RECURRING`).

```
void process_watch_queue(watched_queue *queue, ENABLEDISABLE enable, ONESHOT one_shot);
```

3.6.5 Enqueue an Item onto a Watched Queue

Enqueuing a packet or an arbitrary data pointer places that item at the end of the queue. It also notifies any process watching this queue that a packet has been added to the queue. To enqueue an item, use either the `process_enqueue()` or `process_enqueue_pak()` function:

```
boolean process_enqueue(watched_queuetype *queue, void *whatever);

void process_enqueue_pak(watched_queuetype *queue, paktype *pak);
```

Both functions print an error message if the queue does not exist. If the item cannot be enqueued, `process_enqueue()` returns `FALSE`, whereas `process_enqueue_pak()` returns the packet to the pool of free packets.

3.6.6 Dequeue an Item from a Watched Queue

To remove an item from the head of a managed queue, use the `process_dequeue()` function. If you need to empty a queue, you can call this function from a loop until it returns `NULL`, indicating that the queue is empty

```
void *process_dequeue(queuetype *queue);
```

3.6.7 Locate an Item on the Queue

To return a pointer to the first item on the queue, use the `process_peek_queue()` function.

```
void process_peek_queue(watched_queue *queue);
```

3.6.8 Determine the Size of a Watched Queue

To return the current size of a watched queue, use the `process_queue_size()` function.

```
int process_queue_size(watched_queue *queue);
```

To return the maximum size of a watched queue, use the `process_queue_max_size()` function.

```
int process_queue_max_size(watched_queue *queue);
```

3.6.9 Resize a Watched Queue

To change the maximum allowed size of an existing managed queue, use the `process_queue_resize()` function.

```
void process_queue_resize(watched_queuetype *queue, int new_size, queuetype *overflow);
```

3.6.10 Determine Whether a Queue is Full or Empty

To determine whether a queue is full or empty, use the `process_is_queue_full()` or `process_is_queue_empty()` function.

```
boolean process_is_queue_full(watched_queue *queue);

boolean process_is_queue_empty(watched_queue *queue);
```

3.6.11 Delete a Watched Queue

When a queue is no longer needed, delete it using the `delete_watched_queue()` function. This function unlinks and frees the data structures for any processes that are watching the queue and then frees the queue data structure itself. The function also clears the input parameter to prevent dangling pointers. Make sure the queue is empty when you call the `delete_watched_queue()` function. Otherwise, all elements in the queue are lost.

```
void delete_watched_queue(watched_queue **wq);
```

3.7 Manage Booleans

3.7.1 Boolean: Definition

A *boolean* is a memory location that holds one of two values, TRUE or FALSE (that is, the value 1 or 0, respectively). A *managed boolean* (also called a *watched boolean*) can additionally wake up a process or processes whenever the value of the boolean is set to TRUE (that is, the value 1).

3.7.2 Create a Watched Boolean

Creating a boolean that can be managed by the scheduler allows processes to be automatically scheduled for execution whenever the value of the boolean changes. To create a watched boolean, use the `create_watched_boolean()` function.

```
watched_boolean *create_watched_boolean(char *name, ulong id);
```

3.7.3 Modify the Boolean Minor Type

Use the `process_set_boolean_minor()` function after creating a watched boolean and before watching it to modify the minor type that the boolean receives when the boolean state changes. This function is most useful when the boolean is created in a common function that specifies a default minor identifier for each boolean it creates. The `process_set_boolean_minor()` allows a process to unambiguously re-identify booleans created in this way before watching them.

```
void process_set_boolean_minor(watched_bitfield *event, ulong minor);
```

3.7.4 Set the Value of a Watched Boolean

Setting the value of a managed boolean to TRUE moves all processes watching this variable onto their appropriate processor ready queue if they are not already there. To dothis, use the `process_set_boolean()` function.

```
void process_set_boolean(watched_boolean *wb, boolean value);
```

After the process has been processed, the value of the managed boolean should be set back to FALSE.

A watched boolean event is triggered on a state change from FALSE to TRUE. Setting the boolean if it is already set has no effect, and clearing the boolean also has no effect.

3.7.5 Retrieve the Value of a Watched Boolean

To retrieve the value of a managed boolean, use the `process_get_boolean()` function.

```
boolean process_get_boolean(watched_boolean *wb);
```

3.7.6 Register a Process for Notification on a Watched Boolean

To register a process to be notified that the value of a boolean has changed, use the `process_watch_boolean()` function. In this function, you specify the frequency of the notification; that is, whether the process should be awakened only the first time the boolean changes (ONESHOT) or every time it changes (RECURRING).

```
void process_watch_boolean(watched_boolean *wb, ENABLEDISABLE enable, ONESHOT
one_shot);
```

3.7.7 Delete a Watched Boolean

When a boolean is no longer needed, delete it using the `delete_watched_boolean()` function. This function unlinks and frees the data structures for any processes that are watching this queue and then frees the boolean data structure itself. The function also clears the input parameter to prevent dangling pointers.

```
void delete_watched_boolean(watched_boolean ** wb);
```

3.8 Manage Semaphores

3.8.1 Semaphore: Definition

A *semaphor* is a memory location that is used by multiple processes to serialize their access to a set of resources. The resource can be anything, for example, Flash memory or the table of IP routes. A *simple semaphore* is a single memory location that can be set or cleared by routines that function atomically. It is represented by the basic `semaphore` data structure. A *watched semaphore*, or *managed semaphore*, contains a simple semaphore and all other information necessary so that the semaphore can be used as a scheduler wakeup condition. A watched semaphore is represented by the data type `watched_semaphore`. To guarantee the atomicity of operations that modify a semaphore, you must lock the semaphore before making the modification and unlock the semaphore when the modification is complete.

3.8.2 Create a Watched Semaphore

Creating a semaphore that can be managed by the scheduler allows processes to be scheduled automatically for execution whenever the semaphore is released. To create a watched semaphore, use the `create_watched_semaphore()` function:

```
watched_semaphore *create_watched_semaphore(char * name, ulong id);
```

3.8.3 Modify the Semaphore Minor Type

Use the `process_set_semaphore_minor()` function after creating a watched semaphore and before watching it to modify the minor type that the semaphore receives when the semaphore state changes. This function is most useful when the semaphore is created in a common function that specifies a default minor identifier for each semaphore it creates. The `process_set_semaphore_minor()` allows a process to unambiguously re-identify semaphores created in this way before watching them.

```
void process_set_semaphore_minor(watched_bitfield * event, ulong minor);
```

3.8.4 Lock and Unlock a Semaphore

A semaphore is a memory location that is used by multiple processes to serialize their access to a set of resources. The resource can be anything, for example, Flash memory or the table of IP routes. To guarantee the atomicity of operations that modify a semaphore, you must lock the semaphore before making the modification and unlock the semaphore when the modification is complete.

For a simple semaphore, use the `lock_semaphore()` and `unlock_semaphore()` functions to atomically lock and to unlock the semaphore, respectively.

```
boolean lock_semaphore(semaphore *sem);

void unlock_semaphore(semaphore *sem);
```

For a managed semaphore, use the `process_lock_semaphore()` and `process_unlock_semaphore()` functions to lock and unlock a semaphore, respectively

```
boolean process_lock_semaphore(watched_semaphore *sem, ulong timeout);

boolean process_unlock_semaphore(watched_semaphore *sem);
```

3.8.5 Register a Process for Notification on a Watched Semaphore

To register a process to be notified that a semaphore has been released, use the `process_watch_semaphore()` function. In this function, you specify the frequency of the notification; that is, whether the process should be awakened only the first time the semaphore is released (`ONESHOT`) or every time it is released (`RECURRING`).

```
void process_watch_semaphore(watched_semaphore *sem, ENABLEDISABLE enable,
                             ONESHOT one_shot);
```

3.8.6 Delete a Watched Semaphore

When a semaphore is no longer needed, delete it using the `delete_watched_semaphore()` function. This function unlinks and frees the data structures for any processes that are watching this semaphore, and then frees the semaphore data structure itself. This function also clears the input parameter to prevent dangling pointers.

```
void delete_watched_semaphore(watched_semaphore **ws);
```

3.9 Manage Bit Fields

3.9.1 Bit Fields: Definition

A *bit field* is a 32-bit quantity in which each of the individual bits has significance. This is in contrast with the normal set of bits, such as a long number or an unsigned long number, in which the set is taken as a whole.

3.9.2 Create a Watched Bit Field

Creating a bit field that can be managed by the scheduler allows processes to be awakened automatically whenever the value of the data structure changes. To create a bit field, use the `create_watched_bitfield()` function.

```
watched_bitfield *create_watched_bitfield(char *name, ulong id);
```

3.9.3 Modify the Bit Field Minor Type

Use the `process_set_bitfield_minor()` function after creating a watched bit field and before watching it to modify the minor type that the bit field receives when the bit field state changes. This function is most useful when the bit field is created in a common function that specifies a default minor identifier for each bit field it creates. The `process_set_bitfield_minor()` allows a process to unambiguously re-identify bit fields created in this way before watching them.

```
void process_set_bitfield_minor(watched_bitfield *event, ulong minor);
```

3.9.4 Register a Process for Notification on a Watched Bit Field

To register the process to be notified when the state of a bit field has changed, use the `process_watch_bitfield()` function. In this function, you specify the frequency of the notification; that is, whether the process should be awakened only the first time a bit field changes (`ONESHOT`) or every it changes (`RECURRING`).

```
void process_watch_bitfield(watched_bitfield *wb, ENABLEDISABLE enable,
                            ONESHOT one_shot);
```

3.9.5 Retrieve the Value of a Watched Bit Field

To retrieve the value of a managed bit field, use the `process_get_bitfield()` function.

```
ulong process_get_bitfield(watched_bitfield *wb);
```

3.9.6 Set Bits in a Watched Bit Field

To set only the specified bits in a managed bit field, leaving the other bits unchanged, use the `process_set_bitfield()` function:

```
void process_set_bitfield(watched_bitfield *wb, ulong value, boolean send_wakeup);
```

3.9.7 Clear Bits in a Watched Bit Field

To clear bits in a watched bit field, use the `process_clear_bitfield()` and `process_keep_bitfield()` functions. The `process_clear_bitfield()` function clears the specified bits in a managed bit field, while the `process_keep_bitfield()` function clears all bits except the specified ones.

```
void process_clear_bitfield(watched_bitfield *wb, ulong value, boolean send_wakeup);

void process_keep_bitfield(watched_bitfield *wb, ulong value, boolean send_wakeup);
```

3.9.8 Delete a Watched Bit Field

When a bit field is no longer needed, delete it using the `delete_watched_bitfield()` function. This function unlinks and frees the data structures for any processes that are watching this bit field, and then frees the bit field data structure itself. The function also clears the input parameter to prevent dangling pointers.

```
void delete_watched_bitfield(watched_bitfield **wbf);
```

3.10 Manage Sets of Scheduler Objects

The scheduler provides three functions to manage sets of scheduler objects rather than just specific scheduler objects.

The `process_push_event_list()` function saves the current set of watched events and installs a new set of events. This function performs a complete swap of watched events, not an addition or subtraction as is done with the `process_watch_xxx()` functions. Passing a `NULL` argument to `process_push_event_list()` creates a new empty set of watched events after saving the existing set of events.

```
boolean process_push_event_list(sched_event_set *preexisting);
```

The `process_pop_event_list()` functions serves the opposite function from `process_push_event_list()`. It removes the currents set of events and then restores the previously saved set of watched events. Passing a `NULL` argument to `process_pop_event_list()` discards the list that it removed. Otherwise, this function returns the argument to the caller.

```
boolean process_pop_event_list(sched_event_set **save_current);
```

The `process_push_event_list()` and `process_pop_event_list()` functions are designed to be used by subroutines that do not want to concern themselves with any watched events being used by the caller of the library. If the library calls `process_push_event_list(NULL)` at the very beginning, it can no longer see any of its caller's events nor can those events wake up the process. The library then proceeds to install the events it wants to watch and continues execution. When the library is finished executing, it should clean up the watched events that it installed and then call `process_pop_event_list(NULL)` just before returning to the caller. This restores the caller's set of watched events and leaves the watched event list exactly where it was when the library was first called. Any events for the calling process that occurred between the calls to `process_push_event_list()` and `process_pop_event_list()` are saved and are made available to the process the next time it calls `process_get_wakeup()`.

If a library routine always builds the same set of events, and if the routine is nonreentrant, it can pass arguments to the `process_push_event_list()` and `process_pop_event_list()` functions. This saves the overhead of building the same set of watched events each time the library is called. The library routine can simply installs and de-installs a preexisting set of watched events.

After a process has called the `process_push_event_list()` or `process_pop_event_list()` function, it can determine whether any "hidden" events have occurred by calling the `process_caller_has_events()` function.

```
boolean process_caller_has_events(void);
```

3.11 Scheduler: Example Code

The following example shows code for creating, managing, and exiting from a Banyan VINES router process.

Process Setup

```
process vines_router (void)
{
    ulong major, minor;

    /*
     * Set up this process' world.
     */

    signal_permanent(SIGEXIT, vines_router_teardown);
    vinesrtpQ = create_watched_queue("VINES RTP packets", 0, 0);
    process_watch_queue(vinesrtpQ, ENABLE, RECURRING);
    reg_add_route_adjust_msg(vines_rtr_pid, "vines_router");
    reg_add_media_fr_pvc_active(vines_rtr_pid, "vines_router");
    reg_add_media_fr_pvc_inactive(vines_rtr_pid, "vines_router");
    process_watch_mgd_timer(&vines_timers, ENABLE);
    .
    .
    .
}
```

Main Loop

```
while (TRUE) {

    /*
     * Wait for the next event to occur.
     */

    process_wait_for_event();
    while (process_get_wakeup(&major, &minor)) {
        switch (major) {
        case TIMER_EVENT:
            vroute_do_timers();
            break;
        case QUEUE_EVENT:
            vroute_process_queues();
            break;
        case MESSAGE_EVENT:
            vroute_process_messages();
            break;
        default:
            errmsg(&msgsym(UNEXPECTEDEVENT, SCHED), major, minor);
            break;
        }
    }
}
```

Exit Handler

```
void vines_router_teardown(int signal, int dummy1, void *dummy2, char *dummy3)
{
    paktype *pak;

    process_watch_mgd_timer(&vines_timer, DISABLE);
    process_watch_queue(vinesrtpQ, DISABLE, RECURRING);
    while ((pak = process_dequeue(vinesrtpQ)) != NULL)
        retbuffer(pak);
    delete_watched_queue(&vinesrtpQ);
    reg_delete_route_adjust_msg(vines_rtr_pid);
    reg_delete_media_fr_pvc_active(vines_rtr_pid);
    reg_delete_media_fr_pvc_inactive(vines_rtr_pid);
    vines_rtr_pid = 0;
}
```

Memory Management

This chapter discusses regions, memory pools, and the chunk manager, and it describes the memory management functions in the Cisco IOS code. The last section describes Virtual Memory, new in Release 12.0.

4.1 Overview: Memory Management

To understand memory management in the Cisco IOS software, you must understand three groups of concepts:

- Regions and the Region Manager
- Memory Pools, Memory Pool Manager, and Free Lists
- Chunk Manager

4.1.1 Regions and the Region Manager

A *region* is a contiguous area of the Cisco IOS address space. The Cisco IOS software provides a region manager to organize memory hierarchically so that platform-specific and driver-specific code can declare areas of memory to the kernel and the kernel can determine how much memory is installed or available in a platform.

4.1.2 Memory Pools, Memory Pool Manager, and Free Lists

To allow applications to allocate memory from the various regions, a memory pool manager creates memory pools and manages memory within the regions. The memory pool manager can manage multiple regions within a single memory pool; this allows memory to be reclaimed from various corners of system memory and used when required. The memory pool manager uses free lists to hold different sizes of free memory blocks and requests blocks of a specific size when creating memory pools. Having different sizes of memory blocks minimizes the fragmentation of memory and preserves the ability to supply memory blocks of a specified size.

4.1.3 Chunk Manager

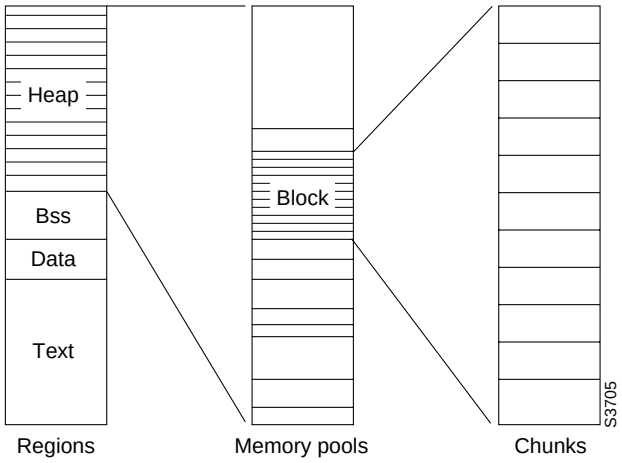
Having the memory pool manager provide comprehensive support for managing areas of memory requires an overhead of context for every block managed. When the memory pool manager is managing many thousands of small blocks, the overhead quickly becomes substantial. To avoid this

overhead, some sections of the Cisco IOS system code allocate a large block of memory called a *chunk*, subdivide it, and manage the subdivided chunks. These chunks are managed by the chunk manager.

4.1.4 Relationship between Regions, Memory Pools, and Chunks

Figur e4-1 illustrates the relationship between regions, memory pools, and chunks. At the left are the regions for a platform. These describe the physical memory in the platform. The heap region is the memory that remains after the image has loaded; the image is represented by the text, data, and BSS regions. The heap normally has a memory pool assigned to it. The memory pool is shown in the middle of the figure and consists of several blocks of memory.These blocks are the areas of memory that are divided up by the allocation code, such as `malloc()`. The blocks can be of different sizes. In the figure, one of these blocks is expanded to show chunks. The block of memory used by the chunk manager looks similar to that of the memory pool, except that chunks consist of identically sized blocks of memory and incur little or no management overhead.

Figure 4-1 Regions, Memory Pools, and Chunks



4.2 Regions

4.2.1 Regions: Definition

A *region* is a contiguous area of the Cisco IOS address space. The Cisco IOS software uses regions to organize memory into a hierarchical and manageable scheme. This organization of memory into regions provides a way for platform-specific and driver-specific code to declare areas of memory to the kernel. It also allows the system code to determine how much memory is installed or available in a platform and where in memory the various sections of an image are located.

In its simplest form, a region is an area of memory that is described by a starting address and a size, in bytes.

A region can also have attributes, such as a class, media access attributes, and inheritance attributes. Region attributes are controlled by the region manager.

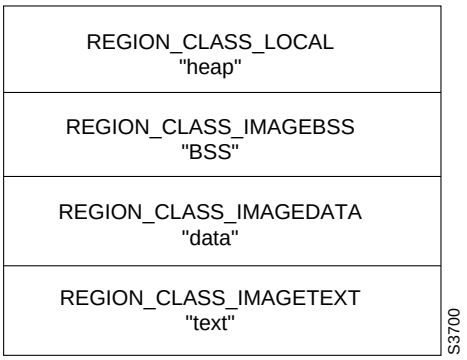
4.2.2 Region Classes: Definition

A *region class* identifies the function for which a region of memory is used. Classes provide a method for organizing regions of memory. Classes have common attributes that allow them to be identified by the region manager and by clients of the region manager.

For example, region classes identify the text, data, BSS, and heap segments of an image's memory.

Figur e4-2 illustrates how a region is organized into classes. In the figure, the *main* region is divided into four regions, which correspond to the text, data, BSS, and heap memory segments.

Figure 4-2 Region Classes



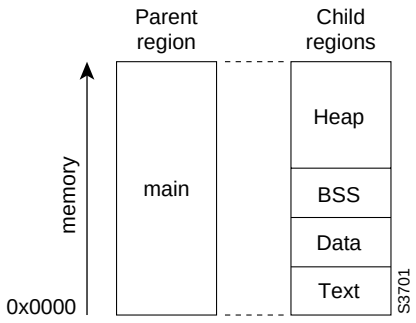
4.2.3 Region Hierarchies: Definition

The Cisco IOS region manager uses region classes to organize memory into a parent-child hierarchy. The region manager creates the hierarchy based on memory base addresses and sizes. The memory allocated to a parent region completely contains the memory allocated to the children of that region. The hierarchy of memory regions can be arbitrarily deep. However, it is normally quite shallow, consisting of two levels, one parent level and one level of children.

Regions are generally declared when a platform is initialized. However, they can be declared to the system at any time. When a region is declared, the region manager arranges the regions in the proper parent-child hierarchy.

Figur e4-3 illustrates how the region manager creates a hierarchy. In the figure, *main* is the parent region of four child regions (*text*, *data*, *BSS*, and *heap*) because the memory allocated to the *main* region is a superset of the memory allocated to all the child regions.

Figure 4-3 Region Hierarchy



Organizing memory regions into a hierarchy is a straightforward way for the system code to determine how much memory is available without counting the memory in overlapping memory areas more than once. For example, the system code needs to know exactly how much main memory is installed on a platform in order to provide output to the user interface, for instance, in response to the **show version** EXEC command.

The hierarchical organization of memory regions also allows the system code to determine where the various sections of an image are located. For example, for subsystem discovery, the system code must be able to locate the data segment. The code can determine this easily because an image's data segment is always a child of the image's *main* memory region and because the data segment is always defined as being in the data region class.

4.2.4 Create a Region

Creating a memory region declares it to the system and automatically registers the region with the region manager. To create a memory region, call the `region_create()` function. In this function, you specify a pointer to the region structure to be initialized, the region's name, starting location and size, class, and any inheritance flags.

```
regi ontype *region_create(regiontype* region, c har*name, v oid*start,
                           ui n size, r e g i o n _cla ss, u i nflags);
```

4.2.4.1 Create a Region: Example

The code in the following example declares the main memory for a standard Cisco platform. The code first interrogates the ROM monitor to determine the system memory size. Then it creates a memory region starting at `RAMBASE` of size `memsize` bytes. The name of the region—*main*—is displayed in the output of user interface commands. The region's class is `REGION_CLASS_LOCAL`, and the region uses the default flags.

```
ulong memsize;

/*
 * Find out how much main DRAM the ROM monitor thinks the system has.
 */
memsize = mon_getmemsize();

/*
 * Add a region to describe all of main DRAM.
 */
region_create(&main_region, "main", RAMBASE, memsize, REGION_CLASS_LOCAL,
              REGION_FLAGS_DEFAULT);
```

In this code example, the region structure is supplied as a pointer to the variable `main_region`. This is important because regions are often created before memory pools are initialized and `malloc()` is available. Most region variables are declared as static variables in the source file that creates them. This allows region creation to be independent of memory pool creation, although the two are usually intrinsically linked.

4.2.5 Set a Region's Class

When you create a region with the `region_create()` function, the region is assigned to the class you specify in the *`class`* parameter. To change a region's class after you have created the region, use the `region_set_class()` function.

```
void region_set_class(regiontype *region, region_class class);
```

4.2.5.1 List of Region Classes

Table 4-1 lists the most common classes used by the system. The region classes that are listed as mandatory in this table are required by the system. Therefore, you must declare at least one region for these classes, either with the `region_create()` or `region_set_class()` function.

Table 4-1 Region Classes

Class	Description
`REGION_CLASS_LOCAL`	(Mandatory) Memory for normal memory and local heaps. This is the default region class.
`REGION_CLASS_IOMEM`	(Optional) Shared memory visible to the processor, network controllers, and other DMA devices.
`REGION_CLASS_FAST`	(Optional) Fast memory, such as SRAM blocks, used for special-purpose and speed-critical tasks.
`REGION_CLASS_IMAGETEXT`	(Mandatory) Region of memory describing the text segment for a running image. This segment contains executable code.
`REGION_CLASS_IMAGEDATA`	(Mandatory) Region of memory describing the data segment. This segment contains all of the initialized variables for an image.
`REGION_CLASS_IMAGEBSS`	(Mandatory) Region of memory describing the BSS segment. This segment contains the uninitialized variables for an image and is zeroed during platform initialization.
`REGION_CLASS_FLASH`	(Optional) Flash memory, which is used by the system for storage. This region is declared primarily on run-from-Flash platforms so that the `REGION_CLASS_IMAGETEXT` regions have valid parents.
`REGION_CLASS_PCIMEM`	(Optional) PCI bus memory, which is visible to all devices on the PCI buses in a platform.

4.2.6 Set Media Access Attributes

A region has media access attributes, which define whether a region is readable and writable. To specify a region's media access attributes, use the `region_set_media()` function.

```
void region_set_media(regiontype *region, region_media media);
```

Calling this function is optional. If you omit it when creating a region with the `region_create()` function, the region is automatically assigned the media type of `REGION_MEDIA_READWRITE`. This media type is appropriate for most regions.

For some regions, you must modify the default media access attributes because of hardware protection issues so that the region media attributes reflect the physical characteristics. For example, text segments are often protected from hardware MMU manipulation. If this is the case, the text segment region should be marked read-only so that applications that are affected by these issues, such as the checksum code, can make proper decisions.

4.2.6.1 List of Media Access Attributes

Table 4-2 lists the possible media access attributes.

Table 4-2 Region Media Access Attributes

Media Access Attribute	Description
REGION_MEDIA_READWRITE	Both read and write operations to the area described by the region are possible. This is the default media access attribute.
REGION_MEDIA_READONLY	Media represented by the region can only be read from.
REGION_MEDIA_WRITEONLY	Media represented by the region can only be written to.
REGION_MEDIA_UNKNOWN	Media access of the region is unknown.
REGION_MEDIA_ANY	(Used in searches only) Find any media access attributes.

4.2.6.2 Example: Media Access Attributes

In the following example, the `region_set_media()` function is used on an R4600-based system to indicate that the text segment is protected against write operations. This code allows the Cisco IOS software to make policy decisions about areas of memory because the Cisco IOS software can determine the memory's vulnerability to erroneous writes.

```
/*
 * Mark the text segment as read only because the R4600 TLB is set up
 * to protect this area.
 */
region_set_media(&text_region, REGION_MEDIA_READONLY);
```

4.2.7 Establish Region Hierarchy

After you have issued the `region_create()` function to create a region, the region manager arranges regions into a hierarchy based on the memory regions' base addresses and sizes, and realigns parent-child relationships if necessary. The hierarchy consists of one or more parent regions and, optionally, child regions of each parent.

4.2.7.1 Region HierarchyTypes

Table 4-3 lists the region hierarchy types that apply to regions.

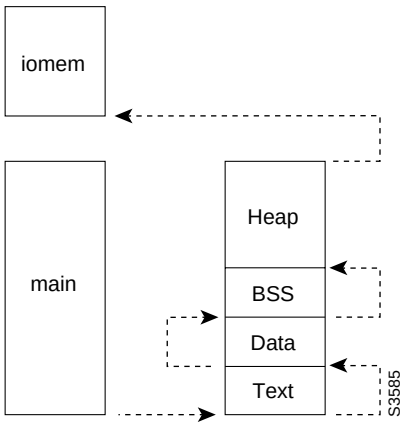
Table 4-3 Region Hierarchy Types

Type	Description
REGION_STATUS_PARENT	Parent region. These memory regions are completely unenclosed b the memory of any other regions. All parents are peers.
REGION_STATUS_CHILD	Child region. These memory regions are enclosed by at least one other memory region.
REGION_STATUS_ALIAS	Alias region. These allow regions to be duplicated at other memory addresses such that the duplicate region does not appear to be a parent.
REGION_STATUS_ANY	(Used in searches only) Finds any region regardless of type.

4.2.7.2 Region Hierarchy Example

Figur e4-4 illustrates a common region hierarchy for a Cisco platform. This figure shows two parent regions, *main* and *iomem*. The *main* region is a parent because it contains other regions—*text*, *data*, *BSS*, and *heap*. These four regions are the children of the *main* region. The *iomem* region is a parent region with no children. The dotted lines in the figure reflect the order of the regions within the region manager

Figure 4-4 main and iomem Region Hierarchy



4.2.8 Establish an Alias Region

The region hierarchy status type `REGION_STATUS_ALIAS` allows regions of memory that are aliased to other, already-declared areas of memory to be accurately reported. Systems where memory addresses signify particular cache policies or byte-swapping manipulation commonly use aliased memory.

To understand why you might want to declare regions as aliases, consider how you would calculate the total size of a particular class of region.To do this, you sum all the parent regions for that particular class. Aliased memory almost always has parent status, so its size is included in the memory total, resulting in an incorrect size. By declaring regions as aliases, they are not included in the sum and their true relationship to the rest of memory is preserved.

To declare a region as an alias, use the `region_add_alias()` function.

```
boolean region_add_alias(regiontype *region, regiontype *alias);
```

4.2.8.1 Example: Establish an Alias Region

The following example of `region_add_alias()` is from the R4600 code. This code declares `k0_main_region` to be an alias of the `main_region` memory region. Aliasing is necessary because the R4600 K0 segment shadows the normal virtual memory map used on R4600-based platforms.

```
region_add_alias(&main_region, &k0_main_region);
```

4.2.9 Set Inheritance Attributes

All regions have inheritance properties associated with them. These properties are passed from a parent region to a child region when the child region is created. You set a regions's inheritance properties using the *flags* parameter of `region_create()` when you create the region.

Whenever the `region_create()`, `region_get_media()`, or `region_set_class()` function is called, the region manager evaluates the region hierarchy to determine whether the attributes of any child regions need to change based on their parent's new attributes.

4.2.9.1 List of Region Inheritance Flags

Table 4-4 lists the possible region inheritance flags.

Table 4-4 Region Inheritance Flags

Inheritance Flag	Description
`REGION_FLAGS_DEFAULT`	Inherit the parent's media type only. Do not modify the child region' class.
`REGION_FLAGS_INHERIT_MEDIA`	Always inherit the parent's media type.
`REGION_FLAGS_INHERIT_CLASS`	Always inherit the parent's region class.

4.2.10 Search through Memory Regions

One of the key reasons for declaring memory to the system is to allow clients of the region manager to search through the known regions for particular memory classes or identify whether a memory address is valid for a particular platform. The following are the main functions for searching for particular memory regions:

```
regiontype *region_find_by_addr(void *address);

regiontype *region_find_by_attributes(region_class class,
                                      region_media media);

regiontype *region_find_next_by_attributes(regiontype* region, region_class class,
                                           region_status status, uregion_media media);

regiontype *region_find_by_class(region_class class);

regiontype *region_find_next_by_class(regiontype *region, region_class class);
```

Calling the `region_find_by_class()` function is equivalent to calling `region_find_by_attributes()` with `REGION_MEDIA_ANY` and `REGION_STATUS_ANY` specified for `media` and `status`, respectively. The same relationship exists between the `region_find_next_by_class()` and the `region_find_next_by_attributes()` functions.

4.2.10.1 Example: Search through Memory Regions by Address

User code often needs to associate region attributes with a given memory address. This can be done with the `region_find_by_addr()` function. The following example illustrates how to associate region attributes with a memory address. This example returns `TRUE` if an address lies within a known text segment (that is, in the `REGION_CLASS_IMAGETEXT` class).

```
boolean is_valid_pc(void *pc)
{
    regiontype *region;

    /*
     * Find the region associated with pc.
     */
    region = region_find_by_addr(pc);
    return(region_get_class(region) == REGION_CLASS_IMAGETEXT));
}
```

4.2.10.2 Example: Search through Memory Regions by All Attributes

It is possible to search the available regions based solely on region class or on all the possible attributes—that is, class, media and status. You can construct loops using the region functions, as shown in the following example:

```
regiontype *region;

/*
 * Find the first data segment region.
 */
region = region_find_by_class(REGION_CLASS_IMAGEDATA);
while (region) {
    ...
    region = region_find_next_by_class(region, REGION_CLASS_IMAGEDATA);
}
```

4.2.11 Determine Whether a Region Class Exists

To determine whether a region class exists, use the `region_exists()` function. If the region exists, this function returns `TRUE`.

```
boolean region_exists(region_class class);
```

4.2.12 Determine a Region's Size

You commonly need to determine the total sizes of various memory region classes. The `region_get_size_by_class()` and `region_get_size_by_attributes()` functions allow you to total the sizes of all regions of the same class and all regions with the same attributes.

```
uint region_get_size_by_class(region_class class);

uint region_get_size_by_attributes(region_class class,
                                   region_media media,
```

Calling the `region_get_size_by_class()` function is equivalent to calling `region_get_size_by_attributes()` for a given class with `REGION_MEDIA_ANY` and `REGION_STATUS_PARENT` specified for `media` and `status`, respectively.

4.2.12.1 Example: Determine a Region's Size

The following example returns the amount of main memory (that is, size of the `REGION_CLASS_LOCAL` class) installed in a platform:

```
uint size;

/*
 * Find the amount of main DRAM installed.
 */
size = region_get_size_by_class(REGION_CLASS_LOCAL);
```

4.2.13 Retrieve a Region's Attributes

Several functions allow you to manipulate a region's attributes.

Note Use the wrapper functiosn described in this section to manipulate a region's attributes. Do not directly manipulate the region attributes in the `regiontype` structure. If you do, the `region_set_class()` and `region_set_media()` functions will not be able to properly track the region's inheritance properties.

To retrieve a region's attributes, use the following functions:

```
region_class region_get_class(regiontype *region);

region_media region_get_media(regiontype *region);

region_status region_get_status(regiontype *region);
```

Table 4-5 summarizes the functions that set and retrieve a region's attributes.

Table 4-5 Set and Retrieve Information about a Region's Attributes

Information about Region	Function to Set	Function to Retrieve
Class	`region_set_class()`	`region_get_class()`
Media	`region_set_media()`	`region_get_media()`
Status	—	`region_get_status()`

Although the `region_get_status()` function exists, `region_set_status()` does not. This is to prevent the region hierarchy from being corrupted by inappropriate manipulation. The only status type that you can set directly is `REGION_STATUS_ALIAS`, which you set by calling `region_add_alias()`.

4.3 Memory Pools

4.3.1 Overview: Memory Pools

The Cisco IOS system code provides memory pools for managing heaps. The region manager supplies regions to the memory pool manager as candidates for memory pools. The memory pool manager creates memory pools to associate disjointed regions into the same pool and hence the same heap segment. Associating disjointed memory areas into a single memory pool allows memory to be reclaimed from various corners of the system memory map and used if required.

The Cisco IOS memory pool support has been designed with the network device environment in mind, because these environments are unlike other common kernel environments, such as the UNI environment. One major difference is that network devices have a variety of memory areas installed on them, and each area has its own properties. The Cisco IOS memory pools make it easy to implement specific network-related requirements such as dynamic memory pool aliasing and alternate memory pools.

4.3.2 Free Lists: Overvie

The free lists used by the Cisco IOS memory pools are different from those used in UNIX and other kernel environments. In these environments, memory managers commonly allocate fixed-size blocks only. This can be inefficient over prolonged periods of time and when used with network devices. To allow the efficient use and re-use of memory, the Cisco IOS memory pool manager uses threaded lists of similarly sized free memory blocks to hold any available memory.

Each free list has a particular size associated with it. Each free block on a list has a size that is equal to or greater than the free list size (but obviously not greater than the next larger free list size). When looking for an available block of memory during a `malloc()` call, the memory manager looks for a block of memory on the free list that is the best fit for the size requested. If no blocks exist on the best-fit free list, the memory manager uses the next higher free list and fragments the block. The fragment is then requeued on a suitably sized (and usually much smaller) free list. This method attempts to find blocks quickly and with minimal fragmentation.

when an allocated memory block is returned to a pool, the memory manager attempts to coalesce physically adjacent blocks to form larger blocks. If this happens, the newly coalesced block is moved to a free list of the correct size. The Cisco IOS software performs no background coalescing and heap cleanup; all coalescing operations happen during the `free()` call.

When you create a memory pool, you can specify a list of initial memory pool free list sizes in bytes or you can use the default list. The default free list contains the following memory sizes in bytes: 24, 84, 144, 204, 264, 324, 384, 444, 1500, 2000, 3000, 5000, 10000, 20000, 32768, 65536, 131072, and 262144.

Users who call `malloc()` and `free()` frequently can register their most active and dynamic memory pool sizes to try to alleviate fragmentation and increase the efficiency of the memory pool. The reason for registering memory pool sizes is that certain heavy users of a memory pool can use memory in patterns that result in excessive fragmentation. For example, users who allocate two identically sized blocks at a time and almost immediately hand one back can excessively fragment larger pools if the free list sizes are poorly chosen in that particular area. Allowing a new free list to be created for a common size leads to faster memory allocation and less fragmentation.

4.3.3 Create a Memory Pool

Memory pools are usually created when the system is initialized by calling the `mempool_create()` function. Creating the memory pool registers it automatically for the class specified in the *`class`* parameter, and the pool immediately becomes available for operations by the `malloc()` and `free()` functions.

```
mempool *mempool_create(mempool *mempool, char *name, regiontype *region,
                        ulong alignment, ulong *free_list_sizes,
                        ulong free_list_count, mempool_class class);
```

4.3.3.1 Example: Create a Memory Pool

The following example illustrates how to create a memory pool. First, a region that starts at `heapstart` and has a size of `heapsize` is created. The `MEMPOOL_CLASS_LOCAL` memory pool is then created from this region of memory. The name of the memory pool is "Processor." This name is displayed in any console output. The alignment is given as 0, which means that the default alignment for the processor is used; this alignment is usually `longword`. The parameters *`free_list_sizes`* and *`free_list_count`* are given as `NULL` and 0, respectively. This means that the memory pool is created with the default free list sizes for heap management.

```
/*
 * Declare a region starting at heapstart of heapsize bytes,
 * and create a "local" memory pool based on it.
 */
region_create(&pmem_region, "heap", heapstart, heapsize, REGION_CLASS_LOCAL,
              REGION_FLAGS_DEFAULT);

mempool_create(&pmem_mempool, "Processor", &pmem_region, 0, NULL, 0,
               MEMPOOL_CLASS_LOCAL);
```

The structure for `pmem_mempool` is passed into `mempool_create()` because it is not possible to allocate memory before a memory pool is created. In fact, all memory pools are usually created in static variables declared in the function that is creating the memory pools. This prevents any timing problems when creating memory pools.

4.3.4 Add Regions to a Memory Pool

After a memory pool is created, regions can be added to it to allow a memory pool to span several disjointed areas of memory. New regions are added with the `mempool_add_region()` function.

```
boolean mempool_add_region(mempool *pool, regiontype *region);
```

When adding regions to a memory pool, once a region is assigned to a memory pool, it cannot be removed or deleted from that memory pool.

4.3.5 Set a Memory Pool's Class

When you create a memory pool with the `mempool_create()` function, you assign a class to that pool. The class you choose reflects both the physical media characteristics of the memory being managed and the abstract uses for the memory pool.

4.3.5.1 Mandatory Memory Pool Classes

Some memory pool classes are mandatory; they are required by the system code. This means that at least one memory pool must be declared for these classes.

4.3.5.2 Aliasable Memory Pool Classes

Some memory pool classes are aliasable. This means that they can be aliased for a particular platform.

4.3.5.3 List of Memory Pool Classes

Table 4-6 lists the most common memory pool classes used by the system code.

Table 4-6 Memory Pool Classes

Memory Pool	Description
`MEMPOOL_CLASS_LOCAL`	(Mandatory) Main system heap.
`MEMPOOL_CLASS_IOMEM`	(Mandatory/aliasable) Shared memory for buffer data and controller descriptor blocks.
`MEMPOOL_CLASS_FAST`	(Mandatory/aliasable) Fast memory (defined by the particular platform) for speed critical structures.
`MEMPOOL_CLASS_PSTACK`	(Mandatory/aliasable) Memory for allocating process stacks.
`MEMPOOL_CLASS_ISTACK`	(Mandatory/aliasable) Memory for allocating interrupt stacks.
`MEMPOOL_CLASS_MULTIBUS`	(Optional) Multibus memory present on some older platforms and used as a fallback pool.
`MEMPOOL_CLASS_PCIMEM`	(Optional) PCI memory present on some platforms.

4.3.6 Alias Memory Pools

Not all platforms need to allocate memory for each class of memory pool. For example, many platforms do not have “fast” areas of memory available for speed-critical memory demands. On these platforms, the main system heap is used for these purposes. In these cases, you can alias memory pools to give the illusion of providing support for a mandatory memory pool class. To alias a memory pool, use the `mempool_add_alias_pool()` function.

```
boolean mempool_add_alias_pool(mempool_class class, mempool *alias);
```

4.3.6.1 Example: Alias Memory Pools

In this example, which illustrates how to alias a memory pool, a platform has neither shared nor fast memory. These mandatory memory pools are aliased to point at `pmem_mempool`, which is the name commonly used for the main heap. This example also illustrates that there are no special stack considerations for either of the memory pools. These pools can also be aliased to the main memory heap. Using aliases in this manner allows efficient and flexible use of the available memory.

```
/*
 * Add aliases for mandatory memory pools.
 */
mempool_add_alias_pool(MEMPOOL_CLASS_LOCAL, &pmem_mempool);
mempool_add_alias_pool(MEMPOOL_CLASS_FAST, &pmem_mempool);
mempool_add_alias_pool(MEMPOOL_CLASS_ISTACK, &pmem_mempool);
mempool_add_alias_pool(MEMPOOL_CLASS_PSTACK, &pmem_mempool);
```

4.3.7 Create Alternate Memory Pools

Alternate pools provide a fallback resource when a memory pool runs out of memory. Including alternate pools in the memory pool management design means that the memory pool manager can construct specialized and optimal memory pools frugally, because larger and more general pools are available as a fallback.

To specify alternate pools, use the `mempool_add_alternate_pool()` function.

```
boolean mempool_add_alternate_pool(mempool *pool, mempool *alternate);
```

4.3.7.1 Example: Create Alternate Memory Pools

The `mempool_add_alternate_pool()` function is commonly used to provide a fallback pool for the `MEMPOOL_CLASS_FAST` memory pool, as shown in the following example:

```
/*
 * If fast memory runs out, fall back on the system heap.
 */
mempool_add_alternate_pool(&fast_mempool, &pmem_mempool);
```

4.3.8 Allocate Memor

To allocate memory to a memory pool, you use a family of `malloc()` functions. See Table 4-7 for a list of the `malloc()` functions that points out the unique feature of each function.

All allocated memory can be returned using the `free()` function.

4.3.8.1 Allocate Unaligned Memory

To allocate memory with the default alignment for the pool, use the following functions:

```
void *malloc(uint size);
void *malloc_fast(uint size);
void *malloc_iomem(uint size);
void *malloc_named(uint size, const char *name);
void *malloc_named_fast(uint size, const char *name);
void *malloc_named_iomem(uint size, const char *name);
void *malloc_named_pcimem(uint size, const char *name);
void *malloc_pcimem(uint size);
void *mempool_malloc(mempool_class class, uint size);
```

4.3.8.2 Allocate Aligned Memory

Occasionally, you must allocate memory that has a specified alignment. To do this, use the `malloc_aligned()`, `malloc_iomem_aligned()`, `malloc_named_aligned()`, or `mempool_aligned_malloc()` function, specifying the alignment in bytes.

```
void *malloc_aligned(uint size, uint alignment);
void *malloc_iomem_aligned(uint size, uint alignment);
void *malloc_named_aligned(uint size, const char *name, uint alignment);
void *malloc_named_iomem_aligned(uint size, const char *name, uint alignment);
void *malloc_named_pcimem_aligned(uint size, const char *name, uint alignment);
void *mempool_aligned_malloc(mempool_class class, uint size, uint alignment);
```

4.3.8.3 Comparison of Memory Allocation Functions

Table 4-7 compares the `malloc()` family of functions available for allocating memory.

Table 4-7 malloc() Family of Functions

Description	Function to Allocate General Memory	Function to Allocate Aligned Memory
Allocates memory from `MEMPOOL_CLASS_LOCAL`.	`malloc()`	`malloc_aligned()`
Allocates memory from `MEMPOOL_CLASS_FAST`.	`malloc_fast()`	—
Allocates memory from `MEMPOOL_CLASS_IOMEM`.	`malloc_iomem()`	`malloc_iomem_aligned()`
Allocates memory from `MEMPOOL_CLASS_LOCAL` and binds a textual name to the allocated memory.	`malloc_named()`	`malloc_named_aligned()`
Allocates memory from `MEMPOOL_CLASS_FAST` and binds a textual name to the allocated memory.	`malloc_named_fast()`	—
Allocates memory from `MEMPOOL_CLASS_IOMEM` and binds a textual name to the allocated memory.	`malloc_named_iomem()`	`malloc_named_iomem_aligned()`
Allocates memory from `MEMPOOL_CLASS_PCIMEM` and binds a textual name to the allocated memory.	`malloc_named_pcimem()`	`malloc_named_pcimem_aligned()`
Allocates memory from `MEMPOOL_CLASS_PCIMEM`.	`malloc_pcimem()`	—
General-purpose method of allocating memory. When you call this function, you must specify the memory pool class to which you are allocating memory.	`mempool_malloc()`	`mempool_aligned_malloc()`

4.3.8.4 Guidelines for Allocating Memory

Keep these guidelines in mind when designing code that allocates memory:

- Always check the return value of all calls to the `malloc()` family of functions.

  Check the return value for a return code of `NULL`, which indicates that no memory is available. Unlike UNIX systems or other virtual memory platforms, embedded systems such as network devices can run out of memory. Failure to check the return code of a `malloc()` request can result in memory corruption in systems that do not have MMU support or protection.

  As the Cisco IOS system image grows larger, the heap size gets squeezed and network platforms run out of memory. Code such as the following fragment dereferences `NULL` if `malloc()` fails:

  ```
  ptr = malloc(sizeof(snark_t));
  ptr->flags = SNARK_DEFAULT_FLAGS;
  ```

  On 68000-based platforms, this code makes the values in low memory unusable. For some releases, the exception table is located in low memory. This type of problem is difficult to debug. On R4600-based platforms, this exception is caught and the offending party recorded as part of the crash. In either case, this is a catastrophic bug that is difficult to trace at a customer site.

- A call to `malloc()` does *not* need a cast.

  All the memory allocation functions return a type of `void *`. Therefore, no typecasting is required. This allows the code to be cleaner than code littered with typecasts that subvert any typechecking that the compiler can perform. Code such as the following is completely spurious:

  ```
  ptr = (snark_t *)malloc(sizeof(snark_t));
  ```

  Also, casts effectively circumvent any type checking that **gcc** can perform.

- Failures in `malloc()` are recorded by the memory management code.

  In earlier Cisco IOS software releases, the following type of code was common:

```
ptr = malloc(sizeof(snark_t));
if (!ptr) {
    errmsg(&msgsym(NOMEMORY, SNARK), “snark structure”);
    return;
}
```

  However, this resulted in a huge amount of text segment space being used by these `errmsg()` (or sometimes `buginf()`) calls. Starting with Software Release 11.0, no error logging should be produced locally by `malloc()` failures. Instead, the `malloc()` function produces rate-limited error messages of the following form:

```
Jun 17 16:58:37: %SYS-2-MALLOCFAIL: Memory allocation of 16777236 bytes failed
  from 0x3E81E, pool Processor, alignment 0
-Process= “Exec”, ipl= 0, pid= 42
-Traceback= E3FE F3D0 3E826 3EE5E 6361E 1CEC4 6C64E
```

  To display the last ten failures, use the **show memory failures allocation** EXEC command:

```
Router# show memory failures allocation
Caller       Pool          Size   Alignment   When
0x3E81E     Processor   16777236      0      0:02:28
0x3E81E     Processor   16777236      0      0:02:26
0x3E81E     Processor   16777236      0      0:02:25
0x3E81E     Processor   16777236      0      0:02:24
0x3E81E     Processor   16777236      0      0:02:23
0x3E81E     Processor   16777236      0      0:02:09
0x3E81E     Processor   16777236      0      0:02:07
0x3E81E     Processor   16777236      0      0:02:07
0x3E81E     Processor   16777236      0      0:02:06
0x3E81E     Processor   16777236      0      0:00:10
```

  The only sections of code that should generate any visible messages are those called by the parser to let the user know the command has failed because of memory shortages. In these cases, the following type of code is acceptable. Note that `printf()` should be used only when the code is being executed by the parser; the string should be the globally provided string `nomemor` .

```
ptr = malloc(sizeof(frobnitz_t));
if (!ptr) {
    printf(nomemory);
    return;
}
```

- For major structures that might consume a large amount of memory, consider calling `named_malloc()` so that the **show memory** EXEC command shows both the name of the allocated memory and what allocated the memory.

- If you choose to locate a `mgd_timer` structure inside of memory that has been allocated using `malloc()`, you must stop the `mgd_timer` before freeing the space. Otherwise, router crashes can occur with `mgd_timer_set_exptime_internal()` in the backtrace. For example, if you call `malloc(sizeof mgd_timer)`, use the `mgd_timer` in that malloc'd space, and then, in error, manage to call `free()` on the space while the timer is still running, the next time some code in the same process calls `mgd_timer_start()`, the router will crash in the `mgd_timer_set_exptime_internal()` routine when it dereferences a n w-poisoned pointer in the `mgd_timer` tree for that process.

  Because it is harmless to call `mgd_timer_stop()` on a timer that is already stopped, you should always call `mgd_timer_stop()` on the timer before calling `free()` for the memory area.

4.3.8.5 Example: Allocate Memory

The following code fragment illustrates how to allocate general memory. This code attempts to allocate memory for two structures. The first attempt uses the `fast_malloc()` function to allocate memory from the `MEMPOOL_CLASS_FAST` memory pool. If the first attempt does not succeed, the code returns a value of `NULL`, which indicates that no memory is available. The second attempt to allocate memory uses `malloc()` to obtain memory from the `MEMPOOL_CLASS_LOCAL` memory pool. If this attempt fails, the code calls `free()` to return the previous allocation and then returns a value of `NULL`.

```
hwidb = malloc_fast(sizeof(hwidbtype));
if (!hwidb)
    return(NULL);

idb = malloc(sizeof(idbtype));
if (!idb) {
    free(hwidb);
    return(NULL);
}
```

4.3.9 Return Memory

To return allocated memory, use the `free()` function.

```
void free(void *memory);
```

4.3.10 Lock and Return Memory

When there are multiple users of a block of memory (such as multiple processes), it often becomes necessary to lock a block so that it is not freed until every user has signalled that they are finished with it. Each block of memory has a reference count associated with it for this purpose. When a block is allocated, it has a reference count (or `refcount`) of 1. To increment the `refcount` for a block of memory, use the `mem_lock()` function.

```
void mem_lock(void *memory);
```

To attempt to return a block of memory, use the `free()` function.You can use `free()` to return all allocated memory.

```
void free(void *memory);
```

If `free()` is called with a block that has a `refcount` of 1, the block is returned to the memory pool from which it was created. If the `refcount` is greater than 1, `free()` decrements `refcount` and returns without doing anything further to the memory block. This mechanism allows any of the potential users of the memory block to be responsible for returning it without risking a memory leak. In this regard, `free()` is the logical equivalent of `mem_unlock()` when using locked blocks of memory.

4.3.10.1 Example: Lock Memory

One of the most common reasons to lock memory is to prevent a block of memory from being freed by another process. Although the Cisco IOS scheduler is a run-to-completion scheduler, there are windows of opportunity for scheduling breaks in areas of the code that do not immediately indicate it. For example, when dumping the contents of a structure that resides in an allocated memory block to a TTY device, it is often important to lock down the block that you are attempting to dump. This is necessary because the user interface routine runs from the context of the EXEC process that handles the TTY device to which a user connects, not the process actively managing the data

structures being displayed. If the TTY device has automore configured, the display process can potentially suspend during any function call that displays text, waiting for the user to allow more output to be displayed.

However, there is usually another active process running in the system that manages these data structures. If, when the display process is suspended, it deletes and frees the structure that the display process was dumping, there will be problems, especially if the structure is in a linked list and contains a pointer to the next element.

To avoid these problems, you can lock the memory block while it is being displayed, as in the following example:

```
while (boojum)
    /*
     * Lock the structure while we are using it.
     */
    mem_lock(boojum);
    /*
     * Display structure information
     */
     .
     .
     .
    /*
     * Unlock structure
     */
    boojumnext = boojum->next;
    free(boojum);
    boojum = boojumnext;
}
```

You must save the `next` pointer before calling `free()`. This is done in case the structure has been freed by the management process while you were displaying it. If that happens, the `free()` function at the end of the display loop physically hands the memory block back because the `refcount` is 1. This is the main reason that there there is no `mem_unlock()` function that effectively calls `free()`. (Many engineers have suggested adding this function.) By using `free()` to unlock memory blocks, the possible side effects of the unlock operation remain immediately obvious.

Do not use fields from the structure after the `free()` is called, because this produces an error that is not immediately caught.

4.3.11 Add Free List Sizes

The default free list contains the following memory sizes in bytes: 24, 84, 144, 204, 264, 324, 384, 444, 1500, 2000, 3000, 5000, 10000, 20000, 32768, 65536, 131072, and 262144.

If you call `malloc()` and `free()` frequently, you can register your most active and dynamic memory pool sizes to try to alleviate fragmentation and increase the efficiency of the memory pool using the `mempool_add_free_list()` function.

```
boolean mempool_add_free_list(mempool_class class, ulong size);
```

4.3.11.1 Example: Add Free List Sizes

The most common time to register memory pool sizes is when subsystems are initialized, as illustrated in the following example. In this example, each call adds a new size to the free list tree for `MEMPOOL_CLASS_LOCAL`, allowing efficient allocation and return of these free list sizes.

```
/*
 * Create some free lists.
 */
mempool_add_free_list(MEMPOOL_CLASS_LOCAL, sizeof(gdbtype));
mempool_add_free_list(MEMPOOL_CLASS_LOCAL, sizeof(mdbtype));
mempool_add_free_list(MEMPOOL_CLASS_LOCAL, sizeof(midbtype));
```

4.3.12 Specify Low-Memory Actions

Memory pool users might need to specify an emergency action to take if available memory becomes too low. In order for these memory pool users to function efficiently, they need an early warning that memory is running low. The Cisco IOS system code provides two thresholds that can be monitored to provide early warning: a low-memory threshold and a fragment threshold.

4.3.12.1 Set the Low-Memory Threshold

The low-memory threshold is triggered when the amount of free memory in a pool drops below a specified amount. The default threshold for the `MEMPOOL_CLASS_LOCAL` memory pool class is 96 KB. Other memory pool classes have no default thresholds. To set or change the low-memory threshold, use the `mempool_set_fragment_threshold()` function.

```
void mempool_set_fragment_threshold(mempool_class class, ulong size);
```

4.3.12.2 Set the Fragment Threshold

The fragment threshold is triggered when the size of the largest block free in a memory pool is smaller than a specified amount. The default threshold for the `MEMPOOL_CLASS_LOCAL` memory pool class is 32 KB. Other memory pool classes have no default thresholds. Toset or change the fragment threshold, call the `mempool_set_low_threshold()` function.

```
void mempool_set_low_threshold(mempool_class class, ulong size);
```

4.3.12.3 Determine Whether Memory Is Low

The `mempool_is_empty()` function returns `TRUE` if the memory pool and its optional alternate have dropped below both the low-memory and fragment thresholds. This is a relatively expensive check because the free lists must be checked for fragment size and the memory pool totals must also be checked.

```
boolean mempool_is_empty(mempool_class class);
```

An alternative and much less CPU-intensive function is `mempool_is_low()`, which checks only the total number of bytes free in the pool against the low threshold if it is set.

```
boolean mempool_is_low(mempool_class class);
```

4.3.13 Search through Memory Pools

To search for particular memory pools by address or class, use the `mempool_find_by_addr()` and `mempool_find_by_class()` functions.

```
mempool *mempool_find_by_addr(void *address);

mempool *mempool_find_by_class(mempool_class class);
```

4.3.13.1 Example: Search through Memory Pools by Memory Pool Address

The following example illustrates how to find the memory pool that manages a given memory address. This example returns `TRUE` if an address is managed by a memory pool.

```
boolean address_is_managed (void *address)
{
    mempool *mempool;

    /*
     * Find the mempool associated with address.
     */
    mempool = mempool_find_by_addr(address);
    return(mempool != NULL);
}
```

4.3.13.2 Example: Search through Memory Pools by Memory Pool Class

The following example searches the available memory pools based on the memory pool class:

```
mempool *fast_mempool;

/*
 * Find the memory pool of fast memory.
 */
fast_mempool = mempool_find_by_class(MEMPOOL_CLASS_FAST);
```

4.3.14 Retrieve Statistics about a Memory Pool

The `mempool_get_free_bytes()`, `mempool_get_total_bytes()`, and `mempool_get_used_bytes()` functions check the current state of a memory pool class and return statistics about memory pools. These functions are used by memory management functions to provide information about the current state of a memory pool class.

```
ulong mempool_get_free_bytes(mempool_class class);

ulong mempool_get_total_bytes(mempool_class class);

ulong mempool_get_used_bytes(mempool_class class);
```

4.4 Chunk Manager

4.4.1 Overview: Chunk Manager

The memory pool manager provides comprehensive support for managing areas of memory. This requires an overhead of context for every block managed, which is usually about 32 bytes per block. If a section of code is to manage many thousands of small blocks, the overhead quickly becomes substantial. To avoid this overhead, some sections of code allocate a large block of memory,

subdivide it into chunks, and manage the subdivided chunks. These chunks are managed by the *chunk manage* . The chunk manager provides a standard method for managing specialized blocks of memory.

In addition to being more efficient for managing small element sizes, using blocks managed by the chunk manager allows the memory pool manager to avoid the fragmentation that results from allocating thousands of small elements from a large memory pool regardless of whether a free list exists for that size.

4.4.2 Guidelines for Using the Chunk Manager

Follow these guidelines when using the chunk manager:

- The blocks allocated from a chunk must all be the same size.
- A chunk can grow dynamically, but it cannot grow from interrupt level. This restriction is a function of how the code for memory pools is allocated. The dynamic chunks are chained to the initial chunk structure, and, if they become full of free elements, they are “trimmed” to prevent bursts of chunk usage from causing long-term memory shortages.
- When creating new chunks, be careful about the number of chunk elements allocated for each chunk. Allocating too many elements per chunk might mean that more memory is wasted by the chunk manager than would be wasted by using the standard memory pool manager. This is because when you use the chunk manager, memory is allocated in one large block and the majority of the block is unused.
- Use the chunk manager only if the probability of memory corruption is low.

4.4.3 Create a Memory Chunk

To allocate a chunk of memory to be managed by the chunk manager, use the `chunk_create()` function. If no memory pool is specified, the chunk is created out of the `MEMPOOL_CLASS_LOCAL` memory pool.

```
chunk_type *chunk_create(uint size, uint maximum, uint flags, mempool *mempool,
                         ulong alignment, char *name);
```

The *`flags`* parameter can be any combination of the flags listed in Table4-8.

Table 4-8 Chunk Pool Flags

Chunk Pool Flags	Description
`CHUNK_FLAGS_DYNAMIC`	Allows the chunk pool to grow automatically if it becomes exhausted. When the chunk pool grows, a new sibling chunk is created using the parameters supplied for the original chunk block and is made available for allocation.
`CHUNK_FLAGS_LOCKABLE`	Allows elements supplied from the chunk pool to be locked by users with a reference count.

Use the **show chunk** command to verify that you are actually using the chunks that you created and that they are the proper size.

4.4.3.1 Example: Create a Memory Chunk

The following example creates a managed memory chunk that has `RDB_CHUNK_MAX` elements per chunk. No memory pool or special alignment is requested. Each element is `sizeof(rdbtype)` bytes long. Setting `CHUNK_FLAGS_DYNAMIC` allows new elements to be allocated, chaining them to the end of `rdb_chunks` if it runs out of free elements. Each new chunk created contains `RDB_CHUNK_MAX` elements.

```
/*
 * Initialize IP route structures.
 */
rdb_chunks = chunk_create(sizeof(rdbtype), RDB_CHUNK_MAX, CHUNK_FLAGS_DYNAMIC,
                          NULL, 0, "IP RDB Chunk");
```

4.4.4 Allocate and Return a Memory Chunk Element

After a chunk has been created, you can allocate elements from it using the `chunk_malloc()` function. To return allocated elements, use the `chunk_free()` function.

```
void *chunk_malloc(chunk_type *chunk);

boolean chunk_free(chunk_type *chunk, void *element);
```

You use these functions similarly to the way you use the analogous memory pool functions. The only difference is that you must always specify the chunk context to be used.

The `chunk_malloc()` function, like `malloc()`, can return `NULL` if no element can be obtained. You must always check the return value from `chunk_malloc()`. Failure to do this can result in memory corruption in systems that do not protect low memory.

All the chunk memory allocation functions return a type of `void *`. Therefore, no typecasting is required, thus allowing for cleaner code.

4.4.4.1 Example: Allocate a Memory Chunk

In the following example, an `rdb` entry is allocated by calling `chunk_malloc()`:

```
rdbtype *rdb;

/*
 * Get an rdb entry.
 */
rdb = chunk_malloc(rdb_chunks);
if (!rdb)
    return;
...
chunk_free(rdb_chunks, rdb);
```

4.4.5 Lock a Memory Chunk

If the chunk pool was created with the `CHUNK_FLAGS_LOCKABLE` flag set, each element in the chunk pool has a reference count associated with it that can be incremented by the `chunk_lock()` function.

```
boolean chunk_lock(chunk_type *chunk, void *element);
```

The locking of chunk elements with `chunk_lock()` is analogous to the `mem_lock()` function for data blocks allocated via `malloc()`, and the same examples and warnings apply.

4.4.6 Destroy a Memory Chunk

Occasionally, you might want to destroy a chunk structure if it is no longer required by the code. To destroy a chunk, call the `chunk_destroy()` function.

```
boolean chunk_destroy(chunk_type *chunk);
```

The chunk is destroyed only if all the elements that belong to the chunk chain have been returned before the destruction attempt. This restriction prevents memory corruption by users of the chunk that hold pointers to the memory returned to the memory manager.

4.5 Memory Management Example

4.5.1 Determine Amount of Memory Available

One reason to organize memory regions into a hierarchy is to allow the system code to easily determine how much memory is available without counting the memory in overlapping memory areas more than once. For example, the system code needs to know exactly how much main memory is installed on a platform in order to provide output to the **show version** EXEC commands. The following example of output from this command shows that 16384 KB of memory are available in the *main* region (this value is the sum of the sizes of all the parent `REGION_CLASS_LOCAL` classes) and 4096 KB are available in the *iomem* region. (This value is the sum of all the parent `REGION_CLASS_IOMEM` classes.)

```
Router# show version

Cisco Internetwork Operating System Software
IOS (tm) 4000 Software (XX-K-M), Experimental Version 11.0(16680) [smackie 138]
Copyright (c) 1986-1995 by cisco Systems, Inc.
Compiled Sun 21-May-95 22:40 by smackie
Image text-base: 0x00012000, data-base: 0x0052A5F8

ROM: System Bootstrap, Version 4.14(7), SOFTWARE

Router uptime is 18 hours, 55 minutes
System restarted by reload
System image file is “smackie/PortReady/xx-k-m”, booted via tftp from 171.69.1.129

cisco 4000 (68030) processor (revision 0xB0) with 16384K/4096K bytes of memory.
Processor board ID 5016716
G.703/E1 software, Version 1.0.
Bridging software.
X.25 software, Version 2.0, NET2, BFE and GOSIP compliant.
2 Ethernet/IEEE 802.3 interfaces.
2 Token Ring/IEEE 802.5 interfaces.
2 Serial network interfaces.
128K bytes of non-volatile configuration memory.
4096K bytes of processor board System flash (Read/Write)
```

The output of the **show region** EXEC command shows the locations and sizes, in bytes, of the memory regions. This command shows that the *main* region is 16777216 bytes. This memory region corresponds to the 16384 KB of memory reported by the **show version** command. The *iomem* region has 4194304 bytes, which corresponds to the 4096 KB reported by the **show version** command.

```
Router# show region

Region Manager:

     Start         End     Size(b)  Class  Media  Name
0x00000000  0x00FFFFFF    16777216  Local  R/W    main
0x00001000  0x00010FFF       65536  Fast   R/W    main:sram
0x00012000  0x0052A5F7     5342712  IText  R/W    main:text
0x0052A5F8  0x00552A9F      165032  IData  R/W    main:data
0x00552AA0  0x005B7B3B      413852  IBss   R/W    main:bss
0x005B7B3C  0x00FFFFFF    10781892  Local  R/W    main:heap
0x03000000  0x033FFFFF     4194304  Flash  R/O    flash
0x06000000  0x063FFFFF     4194304  Iomem  R/W    iomem
```

4.6 Virtual Memory

As of Release 12.0, Cisco IOS software provides support for virtual memory (VM). Cisco IOS VM can "increase" memory by as much as 75% of the raw Cisco IOS image size. VM also extends memory protection safeguards beyond those available on platforms without VM. As such, VM is most suited for very low-end systems and systems that require high reliability.

At the time of this writing (September 1998), VM was in its first "incarnation." A project was already underway to reduce the porting and maintenance effort required by the current .link file and image creation methods. In addition, a more formal, detailed documentation is planned for the next iteration.

This chapter provides the information that you need to get started working with VM:

- Introduction to VM: the "Paging Game," an entertaining but accurate introduction to concepts
- Overview of Cisco IOS VM: benefits and costs of using VM; requirements
- Engineering Effort: changes you will have to make to your platform
- VM Rules: rules that VM "lives by;" advice
- VM Primer: addressing basics
- Porting VM to a Platform: steps
- Wish List: planned improvements
- Style Considerations: list of VM coding and notation style
- Basic VM Terms and Concepts: definitions. Read this first if you are not familiar with VM theory and practice.

4.6.1 Introduction to VM

The Paging Game is a humorous but accurate introduction to virtual memory. It is part of the "Thing King" story, written by Jeff Berryman of the University of British Columbia. The "Thing King" story was distributed at a share meeting shortly after IBM announced virtual memory for the 370 series. See if you can match the players in the Paging Game with real-world VM counterparts.

4.6.1.1 The Paging Game: Rules

1. Each player gets several million “things.”
2. “Things” are kept in “crates” that hold 4096 “things” apiece. “Things” in the same “crate” are called “crate-mates.”
3. “Crates” are stored either in the “workshop” or the “warehouse.” The workshop is almost always too small to hold all the crates.
4. There is only one workshop, but there may be many warehouses. Everybody shares these.
5. To identify things, each thing has its own “thing number.”
6. What you do with a thing is to “zark” it. Everybody takes turns zarking.
7. You can only “zark” your things or shared things, not anyone else’s.
8. Things can only be “zarked” when they are in the workshop.
9. Only the “Thing King” knows whether a thing is in the workshop or the warehouse.
10. The longer the things in a crate go without being zarked, the grubbier the crate is said to become.
11. The way you get things is to ask the “Thing King.” He only gives out things in multiples of 4096 (that is, “crates”). This is to keep the royal overhead down.
12. The way you zark a thing is to give its thing number. If you give the number of a thing that happens to be in the workshop, it gets zarked right away. If it is in a warehouse, the Thing King packs the crate containing your thing into the workshop. If there is no room in the workshop, he first finds the grubbiest crate in the workshop (regardless of whether it is yours or someone else’s) and packs it off (along with its crate-mates) to a warehouse. In its place he puts the crate containing your thing. Your thing then gets zarked, and you never knew that it wasn’t in the workshop all along.
13. Each player’s stock of things has the same thing numbers (to the players) as everyone else’s. The Thing King always knows who owns what thing, and whose turn it is to zark. Thus, one player can never accidentally zark another player’s things, even though they may have the same thing numbers.

4.6.1.2 The Paging Game: Notes

Traditionally, the Thing King sits at a large, segmented table, and is attended by pages, the so-called “table pages,” whose job it is to help the Thing King remember where all the things are and to whom they belong.

One consequence of rule # 13 is that everyone’s thing numbers will be the similar from game to game, regardless of the number of players.

The Thing King has a few things of his own, some of which get grubbier, just as player’s things do, and so move back and forth between the workshop and the warehouse.

4.6.2 Overview of Cisco IOS VM

This section discusses the requirements of VM on a Cisco IOS platform and potential benefits and costs.

4.6.2.1 Requirements

VM requires a platform with an MMU. If your platform does not have one, there is absolutely no way that you can use VM. If you do have an MMU, then you can add VM.

Note Many CPUs have built in MMUs, for example, PPC, MIPS R4k, i386, and others. The older 68k CPUs do not have an MMU.

4.6.2.2 Benefits and Costs

Adding VM to your platform yields two benefits: it decreases the minimum DRAM required to run the Cicso IOS software and it increases quality through improved memory protection beyond that supported by previously protected IOS platforms. The cost is a performance impact, which varies from zero to pretty much as high as you are willing to accept. Returns diminish starting at about 10-15% CPU overhead.

In general, the improved memory protection has negligible impact on performance and offers the following safeguards:

- NULL and illegal address protection: panic on illegal writes, warn on illegal reads. The router can now stay up on NULL pointer reads.
- Free block protection: attempts to read or write memory already freed will warn or panic depending on how VM is initialized.
- Stack overrun protection: panic if a process uses too much stack.
- Stack growth: allocates stack on demand, up to the overrun limit.
- All the normal IOS protections as available on the few platforms that support VM, for example, read-only code sections.

VM "adds" roughly 50% to 75% of the raw image size in DRAM. In other words, if your platform has 16 MB of DRAM with an uncompressed image size of 8 MB, with VM, it would be as if your platform had 20-22 MB of DRAM. This feature does have a performance impact, typically less than 10% overall CPU load.

Since VM only "adds" a fraction of the image size in DRAM, if you have a medium-to-large platform where the amount of physical DRAM is much greater than the image size, it is unlikely that VM will offer a significant DRAM benefit to your platform. For example, adding 8 MB to a 256-MB platform probably will not help much.

Using VM to increase available RAM to the greatest extent possible requires on-unit image storage (in flash, on disk, or in another such resource). You can net-boot a VM image, but your increase in available RAM will be much less. Zero-20% of image size is typical. Also, if you flash-boot, you cannot alter the flash image while the Cisco IOS software is running. This means that if you only have enough flash for one image, your customers must rely on something other than the Cisco IOS software to load images, for example, TinyROM, ROMMON, boot helper, or others.

In summary, VM is primarily useful on very low-end systems or systems that require high quality. It only works on platforms that have an MMU. At best, VM offers an equivalent RAM increase of 50-75% raw image size. The increase comes as the expense of performance and, perhaps, live Cisco IOS flash burn. A typical development effort takes 2 to 3 months.

4.6.3 Engineering Effort

This section lists the things that you need to change on your platform to use VM.

If you are using an MPC8xx (embedded PPC) platform, you are in luck. Nearly all the work has already been done. You can either use the Mantis/c800 code directly or as a reference. If you are using a different CPU/MMU, you need to do the following things:

- Port the VM subsystem: add CPU, MMU, Clock (optional, but very nice if you have a high resolution 32-bit HW clock), and page-in support code for your platform. There are stubs and abundant comments in the VM code to help you. (For this step, you need an ICE or patience and creativity).
- Add VM support to gdb, core-dump, exception, and interrupt handlers. While the volume of work here is rather small, it requires a high degree of precision. (An ICE is helpful, too.)
- Convert your platform to use COFF, DWARF, ELF, or any other BFD-supported object format that supports multiple sections. (If you use a.out, this means you.)
- Create a .link file to produce a VM-format image. This requires fair knowledge of gnu-ld or the ability to acquire it. The Mantis/c800 VM .link file can serve as a seed.
- Create a VM-compression image production script. The Mantis/c800 script can serve here, with only slight modifications. You need to provide new image unpacking code if you are using ROMMON instead of TinyROM.
- Add virtual-to-physical conversion code to all of your device drivers that use DMA. Depending on the flexibility of your platform's code, this may require anything from very little change to a complete driver rewrite.
- Update your flash drivers and/or file system to prevent the Cisco IOS software from overwriting an image currently being used for paging.
- Fix addressing bugs in the Cisco IOS software. If your platform has memory protection already, this number will be very small, perhaps zero. If not, VM will probably uncover something on the order of 20-30 existing bugs. Nearly all will be very simple NULL pointer bugs. Perhaps 2 or 3 will be hard corruption bugs.
- Any flash file system can be made to work. However, performance will suffer if the images in flash are ever discontiguous on anything except page boundaries.
- If you are in the hardware design stage and are considering VM, it is nice, but not required, to have a page of all zeroes and, sometimes, a page of illegal page table entries available. Usually, HW can provide both with no additional cost.

4.6.4 VM Rules

1. Only VM code is allowed to interact with the MMU. This rule includes enabling, disabling, updating, flushing, inspecting, *everything*. Vm cannot be implemented if any other code is interacting with the MMU.
2. All startup code must be in core and marked with `PG_I` in the memory map prior to calling `vm_start()`. Not only is it optional for ROMMON to load non-init and non-pager code into RAM—it might not even fit on all units—but VM will destroy all non-init code and data in RAM to ensure that this rule is followed. By *startup code* is meant all code and data referenced prior to calling `vm_start()`, which should be called as soon as possible before calling `main()`.
3. Your .link file is required to define the symbols _[ben]* for every section referenced in the memory map, where _b* is the least valid virtual address, _e* is least invalid virtual address (_b* <= _e*), _n* is the size in bytes, and <*> is the section name (without a leading dot).

4. *Do not* access data outside the pager sections when writing pager code, including strings for `vm_printf()` messages. Accessing date outside the pager sections will cause an endless page fault loop. Hopefully, you will not ever have to write pager code, but you should be aware of this rule in any case.
5. All VM external symbols and files, whether platform-dependent or not, should be prefixed with `vm_`. The prefix helps people answer the question, "Where the heck is the VM source?"

4.6.5 VM Primer

This section explains some basics about addressing, gives you advice about using VM on a Cisco IOS platform, and provides step-by-step instructions for porting VM to a platform.

4.6.5.1 Virtual Addresses vs. Physical Addresses

A *physical address* is the actual number that you place on the address bus in hardware. This number is limited. The primary motivator behind IBM's initial development of virtual memory back in the late 60s was to allow programs to use more addresses than were actually available in hardware. Such programs are said to run in *virtual space* or *virtual memory.*

Since there are more *virtual addresses* than physical addresses, not all virtual addresses are available at all times. When the CPU references a virtual address that is not in core, an exception is generated (called a *page fault*) and the *pager* is invoked. The pager picks a page that has a physical address (hopefully one that will not be used again in a long while), pages it out, pages in the page that the CPU wants, then continues where it left off.

In a Cisco IOS system, *page-out* only picks read-only and non-dirty read-write pages and simply discards them. In the future, the system may compress dirty read/write pages to DRAM, but it does not do that now. The end result of regular paging is that, while the virtual address space looks just as it would on any Cisco IOS router, the physical address space is completely scrambled.

DMA uses physical addresses. On non-VM Cisco IOS platforms, physical addresses are the same as virtual addresses (or require just a simple mask to convert between the two), so drivers need not worry about a buffer being split in two. On VM Cisco IOS platforms, a physical buffer could be split on any page boundary. Large buffers may be split more than once, and even a 2-byte buffer could be split due to VM's address scrambling.

Your DMA drivers all need to be able to support DMA into completely discontiguous buffers. Full scatter-gather support is ideal for zero performance degradation under VM. If your devices do not support scatter-gather, you either need to provide a special Virtual==Physical IO pool or copy data between a known contiguous region and the normal Cisco IOS buffers.

4.6.5.2 What is an "address interval"?

An *address interval* is simply a range of addresses. Mathematically, an interval is a bounded subset of another, usually well known, set. The subset can be inclusive or exclusive of the boundary values. Please refer to the section Style Considerations for descriptions of interval notation.

Cisco IOS VM sets consist of addresses, either physical or virtual; therefore, our intervals are all subsets of addresses. For example, [0,2^32) typically describes the entire virtual address space for a 32-bit CPU and [0,*N* * 2^20) typically describes the entire physical address space for a system with *N* MB of DRAM.

4.6.5.3 Advice on Using VM

1. Taking the effort to properly tag functions and creating extra sections in your image based on locality of use will significantly enhance the performance of an image when low on RAM. The ideal candidates for tagging: rarely used code (init, parser, error handling, etc.) as well as the converse, heavily used code (ISRs, atomic code, etc.).
2. VM assumes you will be using the VM heap. You can use your own, but that is not recommended. Your unit will fail due to lack of memory when a VM heap system will just run slower, and your unit will run slower (perhaps much slower) when a VM heap system is lightly loaded. The only advantage to not using the VM heap is that performance will not vary much with reference to memory use. If you are sure you do not want the heap, do not link in `vm_heap.o`.
3. Make use of the default RAM limit feature for all non-release versions that you build. This feature was originally intended to test low RAM conditions and stress the pager, but it turned out to be very useful to ensure that the Cisco IOS software does not get too fat to run on older units. That is, you do not need a 4-MB unit to see if your image will run in 4 MB. Just set the default limit to 400h (assuming 4-KB pages).
4. Compiling with -DVM just enables VM section tagging. If your .link file is properly laid out, you can then enable or disable VM at link time just by linking it in or not.

4.6.6 Porting VM to a Platfor

Step Make it link and run using the various stub headers in `vm_port.h` (eg. `vm_mmu_none.h`, `vm_cpu_any.h`, `vm_clock_none.h`, etc.). This will ensure that your .link file is pretty close to being properly set up and that you are indeed linking in all the right files.

The only thing that you can actually do with this image that you could not do before is that you can now use the **vm** command-line interface (CLI) command. It will not do anything yet, but at least you can see that it is there.

The steps you need to take here are as follows:

(a) Edit `vm_port.h` to select the appropriate headers.

(b) Edit your platform `.link` file to add the new sections. See `obj-mpc-c800/c800vm.link` for examples.

(c) Edit your platform makefile and/or `makesubsys.platform` files to include the VM subsystem.

Step Make it link with the actual `.[ch]` files for your CPU and MMU. If your CPU and/or MMU is not already represented in existing code, you will need to write new headers and support modules for your CPU/MMU.

Follow the interfaces described in the none/any files you used in Step 1, but this time the functions must really do what they say they're going to do. Refer to existing CPU/MMU modules if you are in doubt about what to do. They are heavily commented.

The quickest way to create every function that you need is usually to just copy the any/none files to your CPU/MMU files and fill in the blanks. It usually takes less than a day to code the CPU support and a couple of days for the MMU.

When you are done with this step, VM should be ready to run in simulation mode; that is, page from RAM. However, do not do it. You need to complete the next step before you are ready to run VM in simulation mode.

Step Update your driver code to use `vm_v2p()`. This is critical. Your platform almost certainly has devices that require physical addresses and not virtual addresses.You need to alter the drivers for these devices to use `vm_v2p()`, which converts a single virtual address interval into multiple physical address intervals.

You can refer to the c800 platform code for examples. The Ethernet code there is usually a good starting reference.

If you do not update your drivers, your image will probably crash mysteriously very shortly after starting VM. There is no address protection on devices that access physical addresses directly, so you will never know what went wrong.

Step Make VM run in simulation mode. Add a call to `vm_start()` into your startup code before you call `main()` but after exceptions are handled well enough to dump a stack trace. Add appropriate memory map definitions for your platform. See `os/main_c800.c` for examples as well as the comments on `vm/vm_core.c:vm_start()`.

Install as much RAM as your platform can support and run your VM image. This will enable VM in simulation mode; that is, it will page from RAM and use a simple loop for the decompression from flash. This will let you debug all the changes you introduced in Steps 1-3. It will also let you do performance testing.

You should ensure that your platform runs just fine in all respects before you move onto the next step.

Step Teach VM how to find the booted image in flash. Select an appropriate page-in method from those available in `vm_port.h`. You can write your page-in method, but there is not much to gain there unless you have hardware to exploit, for example hardware decompression. More information on this topic will be available later, as it is defined.

Step Build a VM-compressed image.You will need to use one of the standard VM-compression methods supported in `vm_port.h` to compress your image. Then, you will need to package your image in such a way that your ROMMON can load the image into flash. As long as flash contains the data exactly as produced by the VM-compression tool (extra headers and trailers are fine), VM will be able to page the image from flash. More information on this topic will be available after the tool has been defined.

Step Test paging from flash. Load your image and test it. It should work just like it did in Step 4 (simulation mode). The difference is that now it is really paging from flash, so you can measure actual performance.

Step Tune and optimize. If you have a powerful CPU, you can support either a much higher fault rate or a much better compression method than those previously supported. Playing with the delay value in simulation mode will let you determine the acceptable performance limits for your router, which will allow you to decide if they justify adding a new page-in method.

You can also improve performance by introducing additional sections into your .link file. Moving rarely-used code into one section will reduce paging, as will moving frequently used code into one section. Typical candidates: parser code, initialization code, and code that is critical for performance, like fast switching.

4.6.7 Wish List

This section lists improvements that could be made to Cisco IOS VM, time and resources permitting.

1. Periodically check dirty pages to see if they are really changed. That is, writing a zero to something that is naturally zero would mark the page dirty, but it really would not be altered and we really could still page it. Very low likelihood of a gain.

2. Real page-out, that is, compress to RAM buffer. We could do this in software if we made the page-out a low priority background process or if we did it in hardware. We likely are not fast enough to do it on demand in software.
3. Split `vm_core.c` into `vm_pager.c`, `vm_init.c`, and `vm_if.c`. The reasons for a jumbo module no longer apply now that we are using the GNU toolset.
4. Better compression. One of our patent claims outlines a method for enhancing the compression ratio, perhaps significantly, with no impact on decompression rate at run-time. It is fairly easy to do, but would require tool enhancements as well as a new page-in, so it has to wait for a subsequent project.
5. Automatic section placement; that is, generate the final `.link` file with a program instead of by hand. This is quite possible using set intersection techniques on pager sections with user supplied rules for section inclusion and reference. It would take a fair amount of work to do this, though, so it needs to wait for a subsequent project.
6. Automatic section variables generated by `gld`. It would be nice if `gld` could create the `_[ben]` section variables for us. This is not hard to do. (Other linkers do this already).
7. Add support for "execute" permission, that is, `PF_X`. Not all MMUs can do this. For example, `MPC8xx` can only do it by marking all data regions guarded. (This feature would allow us to fault if we try to run from an otherwise valid address in something other than `.text`.)
8. Add support in the MPC MMU page table creation code to use larger page sizes for non-IO locked regions. This would help reduce the table walk overhead, but would require some changes to VM core to ensure that locked initially means locked forever. (We will need to do this anyway to support NP/P compression).
9. Kill off processes that do illegal page-faults instead of panicking. This is not terribly difficult and would be very nice to have.

4.6.8 Style Considerations

1. VM requires some minor style deviations from those in Cisco IOS software. They are permanent and have engineering reasons behind them. Here is a list:
   - No > or >= . (ever)
   - Braces must line up.
   - Use 8-character (hard) tabs.
   - Line length is never to exceed 79 characters.
2. When modifying VM code, it does not matter if you follow the current VM style, use current Cisco IOS style, or another style as long as you do not reformat the code gratuitously. Please consider your reviewer and provide well formatted, well commented code.
3. VM code uses standard mathematical notations, both in the comments and for variable names. Table 4-9 provides a brief description:

Table 4-9 Mathematical Notations in VM Code

Symbol	Meaning
=>	Implies
x:y	Onto mapping (sets, intervals) or ratio (constants)

Symbol	Meaning
{a,b}	A set with elements a,b.
(a,b)	An n-tuple (or record) with variables (fields) a, b, etc.
(li,gi)	Open (exclusive) interval, least invalid < e < greatest invalid
[lv,gv]	Closed (inclusive) interval, least valid <= e <= greatest valid
[lv,li)	Open right interval, least valid <= e < least invalid
(li,gv]	Open left interval, least invalid < e <= greatest valid
<s,n>	Start/extent interval == [s,s+n) (<s,n> is a non-std notation)
iff	If-and-only-if (necessary and sufficient, => and <=)
st.	Such that
sb.	Should be
wrt.	With respect to
{}[]?*	csh style filename globbing(for example, as used to describe VM canonical section naming)

4.6.9 Basic VM Terms and Concepts

In this section, basic VM concepts emerge as the following terms are defined: memory management unit (MMU), virtual memory, paging, physical address, virtual address, demand paging, prepaging, page fault, working set, and least recently used.

Note The following definitions are from the Free Online Dictionary of Computing (http://www.instantweb.com/~foldoc/contents.html). The intention is to provide an introductory discussion of the subject. Some details mentioned in the discussion may differ from those in the Cisco IOS virtual memory implementation.

- memory management unit (MMU)

  A hardware device used to support virtual memory and paging by translating virtual addresses into physical addresses.

  The virtual address space (the range of addresses used by the processor) is divided into pages. The page size is 2^N, usually a few kilobytes. The bottom N bits of the address (the offset within a page) are left unchanged. The upper address bits are the (virtual) page number. The MMU contains a page table, which is indexed, possibly associatively, by the page number. Each page table entry (PTE) gives the physical page number corresponding to the virtual one. This is combined with the page offset to give the complete physical address.

A PTE may also include information about whether the page has been written to, when it was last used (for a least recently used replacement algorithm), what kind of processes (user mode, supervisor mode) may read and write it, and whether it should be cached.

It is possible that no physical memory (RAM) has been allocated to a given virtual page, in which case the MMU will signal a page fault to the CPU. The operating system will then try to find a spare page of RAM and set up a new PTE to map it to the requested virtual address. If no RAM is free, it may be necessary to choose an existing page, using some replacement algorithm, and save it to disk. This is known as paging. There may also be a shortage of PTEs, in which case the OS will have to free one for the new mapping.

In a multitasking system, all processes compete for the use of memory and the MMU. Some memory management architectures allow each process to have its own area or configuration of the page table, with a mechanism to switch between different mappings on a process switch. This means that all processes can have the same virtual address space rather than require load-time relocation.

An MMU also solves the problem of fragmentation of memory. After blocks of memory have been allocated and freed, the free memory may become fragmented (discontinuous) so that the largest contiguous block of free memory may be much smaller than the total amount. With virtual memory, a contiguous range of virtual addresses can be mapped to several non-contiguous blocks of physical memory.

- virtual memory

  The address space available to a process running in a system with a memory management unit (MMU).

  The virtual address space is divided into pages. Each physical address output by the CPU is split into a (virtual) page number (the most significant bits) and an offset within the page (the N least significant bits). Each page thus contains 2^N bytes (or whatever the unit of addressing is).

  The offset is left unchanged and the virtual page number is mapped by the memory management unit (MMU) to a physical page number. This is recombined with the offset to give a physical address - a location in physical memory (RAM).

  Virtual memory is usually much larger than physical memory. Paging allows the excess to be stored on hard disk and copied to RAM as required. This makes it possible to run programs for which the total code plus data size is greater than the amount of RAM available. This is known as *demand paged virtual memor* . A page will be copied from disk to RAM if an attempt is made to access it and it is not already present. This paging is performed automatically by collaboration between the CPU, the MMU, and the operating system kernel. The program is unaware of it.

  The performance of a program depends dramatically on how its memory access pattern interacts with the paging scheme. If accesses exhibit a lot of *locality of reference*, that is, each access tends to be close to previous accesses, the performance will be better than if accesses are randomly distributed over the program's address space, thus requiring more paging.

  In a multitasking system, physical memory may contain pages belonging to several programs. Without demand paging, an OS would need to allocate physical memory for the whole of every active program and its data. Such a system might still use an MMU so that each program could be located at the same virtual address and not require run-time relocation. Thus virtual addressing does not necessarily imply the existence of virtual memory. Similarly, a multitasking system might load the whole program and its data into physical memory when it is to be executed and copy it all out to disk when its timeslice expired. Such swapping does not imply virtual memory and is less efficient than paging.

Some application programs implement virtual memory wholly in software, by translating every virtual memory access into a file access, but efficient virtual memory requires hardware and operating system support.

- paging

  A technique for increasing the memory space available by moving infrequently-used parts of a program's working memory from RAM to a secondary storage medium, usually disk. The unit of transfer is called a *page*.

  A memory management unit (MMU) monitors accesses to memory and splits each address into a page number (the most significant bits) and an offset within that page (the lower bits). It then looks up the page number in its page table. The page may be marked as *paged in* or *paged out*. If it is paged in, the memory access can proceed after translating the virtual address to a physical address. If the requested page is paged out, space must be made for it by paging out some other page, that is, copying it to disk.

  The requested page is then located on the area of the disk allocated for *swap space* and is read back into RAM. The page table is updated to indicate that the page is paged in and its physical address recorded.

  The MMU also records whether a page has been modified since it was last paged in. If it has not been modified then there is no need to copy it back to disk and the space can be reused immediately.

  Paging allows the total memory requirements of all running tasks (possibly just one) to exceed the amount of physical memory, whereas swapping simply allows multiple processes to run concurrently, so long as each process on its own fits within physical memory.

- physical address

  The address presented to a computer's main memory in a virtual memory system, in contrast to the virtual address, which is the address generated by the CPU. A memory management unit (MMU) translates virtual addresses into physical addresses.

- virtual address

  A memory location that is accessed by an application program that is running in a system with virtual memory. Intervening hardware and/or software maps the virtual address to real memory (physical memory). During the course of execution of an application, the same virtual address may be mapped to many different physical addresses as data and programs are paged out and paged in to other locations.

- demand paging

  A kind of virtual memory where a page of memory will be paged in if an attempt has been made to access it and it is not already present in main memory. This normally involves a memory management unit (MMU), which looks up the virtual address in a page map to see if it is paged in. If it is not, the operating system will page it in, update the page map, and restart the failed access. This implies that the processor must be able to recover from and restart a failed memory access or must be suspended while some other mechanism is used to perform the paging.

  Paging in a page may first require some other page to be moved from main memory to disk (paged out) to make room. If this page has not been modified since it was paged in, it can simply be reused without writing it back to disk. This is determined from the modified, or *dirty*, flag bit in the page map. A replacement algorithm or policy is used to select the page to be paged out, often the least recently used (LRU) algorithm.

Prepaging is generally more efficient than demand paging.

- prepaging

  A technique whereby the operating system in a paging virtual memory multitasking environment loads all pages of a process's working set into memory before the process is restarted.

  Under demand paging, a process accesses its working set by page faults every time it is restarted. Under prepaging, the system remembers the pages in each process's working set and loads them into physical memory before restarting the process. Prepaging reduces the page fault rate of reloaded processes and, hence, generally improves CPU efficiency.

- page fault

  In a virtual memory system, an access to a page (block) of memory that is not currently mapped to physical memory. When a page fault occurs, the operating system either fetches the page in from secondary storage, usually disk, if the access was legitimate or reports the access as illegal.

- working set

  The set of all pages used by a process during some time interval.

  The working set frequently consists of a relatively small fraction of a process's total virtual memory pages. While a process's entire working set is in physical memory, the process will run without page faults. If the working set is too large for available physical memory, the process causes frequent page faults.

  In a multitasking environment, the information about which pages are in each process's working set allows the memory management system to improve CPU efficiency by prepaging (sometimes called the *working set model*).

- least recently used

  (LRU) A rule used in a paging system that selects a page to be paged out if it has been used (read or written) less recently than any other page. The same rule may also be used in a cache to select which cache entry to flush.

  This rule is based on temporal locality, the observation that, in general, the page that has not been accessed for longest is least likely to be accessed in the near future.

CHAPTER 5

Pools, Buffers, and Particles

5.1 Buffer Management: Overview

The principle purpose of a router or other network device is to send, receive, and forward packets to and from other network devices. Support for handling these packets is fundamental to the system software.

As network devices have become more widespread and varied, so have the media with which they connect to each other. Each media connection has a maximum transmission unit (MTU) size associated with it. This is the maximum size of any frame that can be transmitted on a particular media. MTU sizes can range from 1500 bytes for Ethernet to over 18 KB for Token Ring. This is a large spread of possible packet sizes, especially when considering that not every packet needs to be full. In fact, frames as small as 64 bytes are common on Ethernet. For the underlying packet support code to be efficient, it must handle this wide range of possible sizes without wasting memory, while at the same time not complicating the underlying code.

The Cisco IOS buffer management code uses pools to manage buffer resources. A generic pool manager allows resources to be managed effectively under various code execution constraints. Pools can be dynamic in size, allowing the number of items within them to grow and shrink on demand so that buffer resources can adapt to match the current packet load. The Cisco IOS buffer management code also provides nominal support for pool caches. These can be used to improve performance in performance-critical sections of code by creating fast lookaside lists for managed pool items.

The majority of the Cisco IOS protocol and application code assumes that the frame data presented to it are contiguous. The default buffer managed by the pool code has a contiguous area of memory for the frame data. In order to permit drivers to support scatter-DMA, the Cisco IOS software also allows frame data to be composed of individual blocks called *particles*. The use of particles is currently the exception rather than the rule in the Cisco IOS code base; before using particles in any code, seek design advice from senior Cisco IOS engineers.

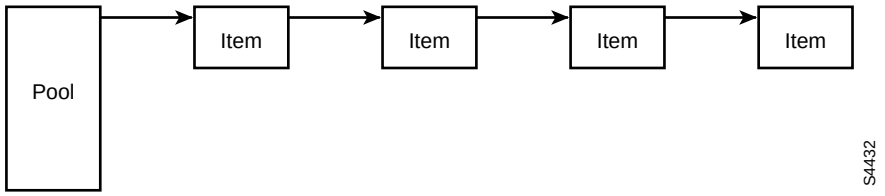
5.2 Generic Pool Management

Many aspects of the Cisco IOS code rely on the management of discrete resources such as buffers and queuing elements. The management of these elements is complicated by the demands placed by the clients of the resource pool. Resources must often be grown and trimmed while preserving the integrity of the pool against a client's running, for example, at interrupt level. This balancing of resources can lead to the duplication of relatively complicated management structures. To simplify this situation, the Cisco IOS software supports generic pool management. All buffer and particle pools are structured on top of the generic pool framework. This framework is open so that other applications can use the core pool support to get the same pool management features as the buffer and particle pools.

5.2.1 Pool Structure

The basic pool architecture is simple (see Figur e5-1). A pool is described by a single `pooltype` structure. From this structure, a queue of pooled items is constructed. All items in the queue are free and ready to be consumed by the pool users. All pool items are threaded together using the usual singly-linked list (queue) support, which means that the first longword of each data item on the queue points to the next item on the queue. The pools themselves are usually threaded together using the list manager support. (Queues and the list manager are discussed in the "Queues and Lists" chapter.)

Figure 5-1 Structure of a Pool



5.2.2 Pool Groups and Size

Two elements of a pool usually dictate its higher-level identification for the client applications of a pool: the size of item to be stored within the pool and the pool's group number.

The size of the item to be stored is a variable component within a group of pools and does not necessarily have to be the full size of the item. For example, in buffer pools, the size indicates the maximum number of bytes available for a network header. The full size of the buffer includes extra constant space for encapsulations and trailers.

Group numbers allow pools to be partitioned from each other while at the same time allowing partial association among pools. For example, three pools could belong to pool group 5 and four could belong to pool group 6. This concept of grouping allows pool users to partition the use of the pools. The pool group number 0 is reserved for public pools.

When a pool is created, a list is required onto which the pool being created can be inserted. Insertion into the list is based on a comparison of the pool size and group number. Pools are arranged in ascending order by group number and, within groups, by size.

5.2.3 Static and Dynamic Pools: Definition

Pools are classified as being either *static* or *dynamic*.

Static pools make no attempt to increase the number of items contained within them if the pool runs low.

With dynamic pools, the pool attempts to meet the demands of its users. If the pool can grow items within the calling context of the caller, it attempts to do so. Otherwise, a critical background process is scheduled to run at the next available interval to fill the pool.

5.2.4 Permanent and Temporary Items: Definition

Items in a pool are classified as being either permanent or temporary.

Permanent items, as their name suggests, are always in the pool and are never destroyed unless the number of permanent items is changed.

Temporary items are transient items that are created in dynamic pools whenever the free count of items in the pool drops below the minimum specified by the `pool_adjust()` function or by the pool user. These items can be removed from the pool if the number of free items rises above the maximum specified.

5.2.5 Create a Pool

To create a pool, use the `pool_create()` function. This function creates only the pool container structure. No items are placed in the pool until it is populated and its parameters defined with the `pool_adjust()` function.

```
pooltype *pool_create (char *name, int group, int size, uint flags, mempool *mempool,
                       list_header *list, pool_item_vectors *item);
```

The key to a pool's operational characteristics is the function vector block passed to it as the *`item`* parameter when the pool is created. This vector operates on an individual pool. It can be one of the values listed in Tabl e5-1. All these vectors are mandatory.

Table 5-1 Pool Item Vectors

Vector	Description
`create`	(Mandatory) Creates and returns a new pool item.
`destroy`	(Mandatory) Destroys a pool item.
`get`	(Mandatory) Gets a pool item from the free queue. This vector can also create a new item if the free queue is empty and the pool is dynamic.
`ret`	(Mandatory) Returns a pool item to the free queue. This vector can also destroy an item if the pool has more than maxfree items in it.
`status`	(Mandatory) Fills in a status block for an item. Item status consists of its state (either temporary or permanent) and its age.
`validate`	(Mandatory) Validates a pool item for debugging support.

The following are the prototypes for the vectors:

```
typedef void * (*pool_item_create_t)(pooltype *pool, pool_item_type type);
ty   pedef void(*pool_item_destroy_t)(pooltype *pool, void *item);
typedef void * (*pool_item_get_t)(pooltype *pool);
ty   pedef void(*pool_item_ret_t)(pooltype *pool, void *item);
ty   pedef void(*pool_item_status_t)(pooltype *pool,
                                     void *item,
                                     pool_item_status *status);
typedef boolean (*pool_item_validate_t)(pooltype *pool, void *item);

typedef struct pool_item_vectors_ {
    po   ol_item_create_tcreate;
    po  ol_item_destroy_tdestroy;
    po      ol_item_get_tget;
    po      ol_item_ret_tret;
    po   ol_item_status_tstatus;
    po ol_item_validate_tvalidate;
} pool_item_vectors;
```

5.2.6 Adjust a Pool

Once a pool has been created, it must be populated before use.To fill a pool and establish the operating parameters for it, use the `pool_adjust()` function. This function sets the requested minimum number of free buffers for the pool, the maximum number of free buffers, and the number of permanent buffers.

```
void pool_adjust(pooltype *pool, int mincount, int maxcount, int permcount,
                 boolean default);
```

Most pool applications use the memory pool manager. (See the "Memory Pools, Memory Pool Manager, and Free Lists" section in the "Memory Management" chapter.) One of the main restrictions of the memory pool manager is that memory cannot be allocated at interrupt level. This effectively means that items cannot be created from an interrupt handler. The pool code, therefore, attempts to keep the minimum specified number of items in the pool whenever it can. If an item is requested from a pool and the free count is below the minimum, the pool code attempts to grow the pool to the minimum free count. The minimum count specified to the pool code can therefore be taken to be the maximum number of buffers that the pool has reserved for interrupt-level requests.

5.3 Pool Caches

5.3.1 Overview: Pool Caches

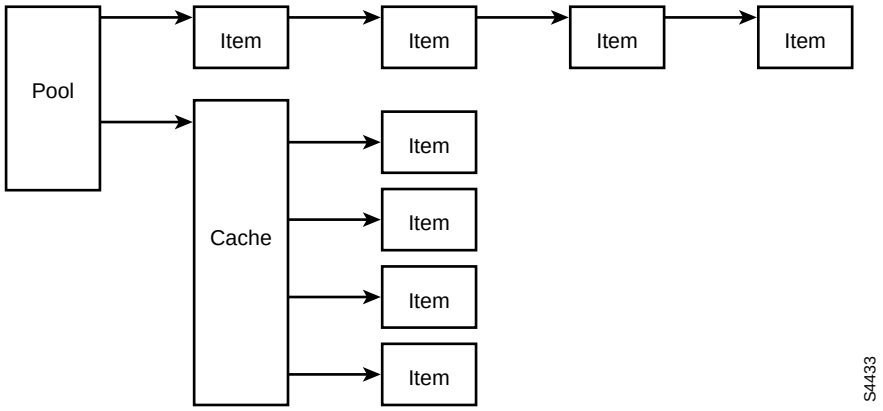
Many of the performance-critical sections of Cisco IOS switching code rely on the pools for supplying various items—either buffers or particles, or both. Allocating items by calling the pool `get` function vector can impose a considerable overhead in paths where every microsecond counts. To reduce this overhead, the pool code provides nominal support for pool item caches. A *pool cache* is effectively a lookaside list of free items that can be accessed quickly.

Pool caches are not transparent, but rather require users of the code to manipulate the fetch from the cache themselves. However, when incorporated into a network driver, a pool cache can provide a noticeable performance boost to critical performance-sensitive paths.

5.3.2 Structure of a Pool with a Cache

Figur e5-2 shows the typical structure of a pool that has a cache. The free list of items, shown across the top of the figure, matches the pool structure shown in Figure5-1 . The cache, shown at the bottom of Figur e5-2, is effectively an array block of pointers. This structure allows the list to be traversed faster, but at the expense of increased pool management risk. You must take great care to ensure that pool cache users have no resource contention or interrupt timing problems.

Figure 5-2 Structure of Pool with a Cache



5.3.3 Add a Pool Cache

To add a pool cache, use the `pool_create_cache()` function. Adding a buffer cache to a pool initializes the structures and internal state required for the pool cache container but does not add an buffers to the cache. To fill the cache, use the `pool_adjust_cache()` function.

```
bool ean pool_create_cache(pooltype* pool, i n maxsize, pool_cache_vectors *cache_item,
                           in  threshold);
```

The pool cache uses a vector block similar to that used by the main pool functions to specify cache policy and cache item creation and deletion. The vectors can be one of the values lis ted in Table5-2. The `create` and `destroy` vectors are mandatory. The `threshold` vector is optional.

Table 5-2 Pool Cache Item Vectors

Vector	Description
`create`	(Mandatory) Creates a new pool cache item. This action usually just fetches an item from the main pool freelist for use in the cache.
`destroy`	(Mandatory) Destroys a pool cache item. This action usually attempt s to return the item to the main pool freelist.
`threshold`	(Optional) Provides flow control management. This vector is called when the pool cache rises above the optional threshold set for it on creation.

The following are the prototypes for the vectors:

```
typedef void * (*pool_cache_get_t)(pooltype *pool);
ty   pedef void(*pool_cache_ret_t)(void *item);
ty   pedef void(*pool_cache_threshold_t)(void);

typedef struct pool_cache_vectors_ {
    po       ol_cache_get_tget;
    po       ol_cache_ret_tret;
    pool_cache_threshold_t threshold;
} pool_cache_vectors;
```

5.3.4 Fill a Pool Cache

After you have created a pool cache with the `pool_create_cache()` function, use the `pool_adjust_cache()` function to fill the cache with items. This function sets the new size of the pool cache.

```
void pool_adjust_cache(pooltype *pool, int new size);
```

Items that are placed into the cache list are not available to any of the `get` vector functions on the pool. Some drivers use this principle to implicitly set aside either buffers or particles for handling incoming traffic to avoid having other users use all the cached items.

Pools that have caches attempt to fill them with any items returned to them. Therefore, no special precautions are required to handle items that have been sourced from a pool cache.

5.3.5 Destroy a Cache

To remove a cache from a pool, use the `pool_destroy()` function:

```
void pool_destroy(pooltype *pool);
```

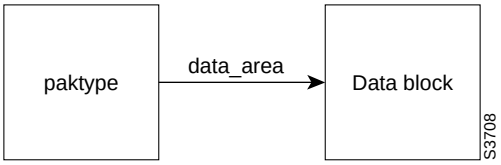
5.4 Buffer Structure

Cisco IOS buffers are split into two sections:

- Buffer Headers
- Buffer Data Blocks

Figur e5-3 illustrates the packet structure.

Figure 5-3 Packet Structure



5.4.1 Buffer Headers

A buffer header contains context and pointers for the data. The header is described by the `paktype` structure. The header usually resides in the main system memory and is usually managed as part of the `MEMPOOL_CLASS_LOCAL` memory pool.

5.4.2 Buffer Data Bloc

A buffer data block contains the packet data. The data is effectively an untyped block of memory that can reside in either the main memory or a shared area of memory, depending on whether the platform uses DMA devices.

All references to the data block are through the `paktype` header. The `data_area` element in this structure points to the base of the data block. Buffer users always pass pointers to `paktype`. They never pass pointers to only the data block.

5.4.2.1 Memory Organization within a Data Block

Within a data block, the memory is usually organized as shown in Figure 5-4. This figure shows the bare minimum of state; far more context is held on the actual buffer.

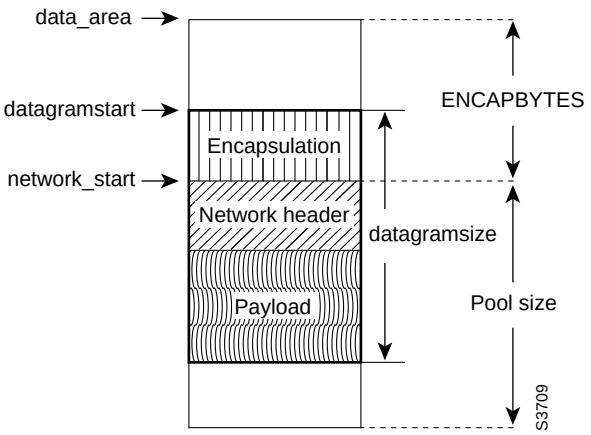
The three shaded areas in the figure indicate a single network frame in the buffer. All frames can be considered as three consecutive pieces in memory:

- An encapsulation (for example, an Ethernet ARPA header)
- A network header (for example, an IP header)
- A payload (for example, a UDP datagram)

The pointer in the buffer header to the start of the complete frame is `datagramstart`. The total size of the frame in memory is `datagramsize`.

The pointer to the start of the network header is `network_start`.

Figure 5-4 Memory Organization within a Data Bloc



In Figure 5-4, the network frame is not located at the start of the data area. This is to allow the encapsulation to grow in size while switching the frame without having to rewrite the network payload. This requirement arises from one of the chief objectives of all high-speed switching paths—that the data should never be accessed more than is required. Copying a payload involves touching almost all of the frame and is very expensive.

The size of the data area is (`ENCAPBYTES` + pool size). The value of `ENCAPBYTES` is defined as the largest encapsulation that the platform supports. The pool size is the variable-sized part of the equation and is usually taken to be the maximum size of a network header and its payload. When dealing with the buffer pool support code, the sizes used are almost always those of the network header and payload because an extra `ENCAPBYTES` is always assumed.

For optimal speed, the code always attempts to place the network header (pointed to by `network_start`) data location `ENCAPBYTES` bytes from the start of the data area. However, there is no guarantee that the network header is placed at this location. Therefore, consumers of a packet should always use `network_start` or other pointers to find the location of the actual data.

5.5 Buffer Pools

5.5.1 Overview: Buffer Pools

The Cisco IOS software uses buffer pools to manage buffers. Buffer pools hold buffers that are the same size and have the same properties. Buffer pools are based on the generic pool manager.

5.5.2 Public and Private Buffer Pools: Definition

Buffers pools are classified as either *public* or *private*.

Public pools are available for everyone to allocate buffers from. All platforms allocate and fill a variety of public buffer pools at run time.

Private pools are primarily used by interface drivers to manage their own MTU-sized pools for incoming traffic. They are visible only to applications with explicit knowledge of them.

5.5.3 Create a Public Buffer Pool

To create a buffer pool, use the `pak_pool_create()` function.

```
pooltype *pak_pool_create(char *name, int group, int size, uint flags, mempool
*mempool);
```

The `pak_pool_create()` function is a wrapper around a call to the `pool_create()` function, with the wrapper specifying the correct vectors for a buffer pool. Public buffer pools all have a group number of 0, which is defined as `POOL_GROUP_PUBLIC`. Different public buffer pools are distinguished from each other by their size. If the size is 0, only buffer headers are created in the pool.

The `pak_pool_create()` function creates a buffer pool only; no buffers are present in the pool. To fill the buffer pool, use the `pool_adjust()` function.

5.5.3.1 Example: Create a Public Buffer Pool

The following example creates a public pool. The pool is called “Large” and has a size of `LARGEDATA` bytes. This means that the size of the data area available for writing network frames is (`ENCA PBYTES+LARGEDATA`) because the size of the maximum encapsulation is always appended. The default flags for the pool are used. This means that the pool is dynamic and that the header and data blocks are from a memory pool. By specifying *`mempool`* as `NULL`, the default memory pool for buffer data is used to source data blocks; this is `MEMPOOL_CLASS_IOMEM`.

```
large = pak_pool_create(“Large”, POOL_GROUP_PUBLIC, LARGEDATA,
                        POOL_DEFAULT_FLAGS, NULL);
```

5.5.4 Create a Private Buffer Pool

When creating a private buffer pool, you must pass a private group ID to the `pool_create()` function. Each private pool has a unique group ID, which is provided via the `pool_create_group()` function.

```
int pool_create_group(void);
```

5.5.4.1 Example: Create a Private Buffer Pool

The following example creates a private buffer pool. This private pool is called "Ethernet," and it contains buffers with enough space for network headers and a payload total of `MAXETHERSIZE` bytes. The pool is not dynamic and has only sanity checking enabled. The data area for each buffer is located in the default `MEMPOOL_CLASS_IOMEM` pool.

```
pool_group = pool_create_group();
buffer_pool = pak_pool_create("Ethernet", pool_group, MAXETHERSIZE, POOL_SANITY, NULL);
```

5.5.5 Obtain a Buffer from a Public Buffer Pool

Most applications in the system image use the public buffer pools as a source of buffers. To obtain a buffer from a public buffer pool, use the `getbuffer()` function.

```
paktype *getbuffer(int size);
```

The *`size`* parameter is the amount of space required for the network header and payload only; each buffer comes with enough space for the largest encapsulation. `getbuffer()` scans all the public buffer pools declares in an attempt to find a pool of the size specified. If the pool that it finds has fewer than the minimum number of free buffers and is dynamic, the buffer code attempts to increase the pool to the minimum value before returning a buffer.

The buffer returned by `getbuffer()` has a variety of pointers already initialized. These include `network_start` and `datagramstart`, which point to the end of the nominal encapsulation area. This means that the pointers can be used immediately to build packets for transmission.

Note Always check the return value from `getbuffer()`. Buffer pools run out, and `getbuffer()` will fail at some point under low-memory or heavy-burst situations in a running system.

5.5.5.1 Example: Obtain a Buffer from a Public Buffer Pool

The following example allocates a buffer from a public pool. In the example, `getbuffer()` requests a packet of size `bytes`. The code then checks whether a buffer exists, and if one does, a pointer to the XNS header to be built is provided via the `XNSHEADSTART` macro. This macro uses `network_start` to point at the XNS header. Once a pointer is found, the header is written into the buffer and the buffer is manipulated before packets are transmitted.

```
pak = getbuffer(bytes);
if (pak == NULL)
    return;
xns = (xnshdrtype *)XNSHEADSTART(pak);
xns->cksum = 0;
xns->len = bytes;
xns->tc = 0;
```

This example is typical of almost all the code that must build buffers using `getbuffer()` or its related functions. Note that `getbuffer()` does not zero the data area before handing a buffer to a user, and all parts of a network header and its payload must be written for each frame.

5.5.6 Obtain a Buffer from a Private Buffer Pool

To obtain a buffer from a private pool, use the `pool_getbuffer()` function. The only parameter supplied to `pool_getbuffer()` is a pointer to the pool from which to obtain the buffer

```
paktype *pool_getbuffer(pooltype *pool);
```

5.5.7 Lock a Buffer

To increment the reference count field of buffer, that is, to indicate that `refcount` is in long-term use and should not be immediately returned, use the `pak_lock` macro.

```
pak_lock(pak)
```

To free the lock, call `datagram_done()` with the buffer pointer

5.5.8 Return a Buffer to a Pool

To return a buffer to a pool, use either the `datagram_done()` or `retbuffer()` function.

```
void datagram_done(paktype *pak);

void retbuffer(paktype *pak);
```

The difference between these two functions is small but significant. In the buffer header, the field `refcount` is a reference count that indicates the number of "users" of a buffer. Incrementing `refcount` using the `pak_lock` macro allows the system code to indicate that the buffer should not be returned until all users have relinquished their hold on the buffer

The `retbuffer()` function expects all buffers passed to it to have a `refcount` value of 1. This is the usual value for all buffers that have been freshly acquired through `getbuffer()`. If a buffer is passed to `retbuffer()` with a `refcount` value greater than 1, an error message is generated.

The `datagram_done()` function can accept any value in `refcount`. If the value of `refcount` is greater than 1, `datagram_done()` decreases the value by 1, and when the value of `refcount` reaches 1, `datagram_done()` returns the buffer. In this sense, `datagram_done()` can be considered to be a type of "`pak_unlock()`" function that is used by applications to relinquish their hold on a buffer, with the last user to let go actually returning the buffer.

In a future release, the functionality of `datagram_done()` will be merged into `retbuffer()`. In the meantime, all applications that return buffers should use `datagram_done()`.

5.5.8.1 Guidelines for Returning a Buffer

Follow these guidelines when designing code to return a buffer:

- Do *not* make decisions based on the reference count, and do *not* add an explicit unlock. If a packet or buffer is being shared, you cannot make assumptions about what the other users are doing or what order the others will be done with the packet. No code should retrieve the reference count except the buffer management code itself.
- Do *not* add any explicit unlocks, or there will be a risk of race conditions. Unlocking and freeing must be integrated.

5.5.9 Duplicate a Buffer

5.5.9.1 Overview: Duplicate a Buffer

Often, you need to duplicate a buffer so that it can be sent to multiple destinations or modified without the risk of changing a buffer that may be waiting in a transmit queue. Copying a frame is the only way to guarantee that its contents are valid. It can take several tens of milliseconds to transmit some frames after sending them to a driver, especially on slow serial links. This means that once a

buffer is handed to a driver, no further manipulation on that buffer can take place. The only way to asynchronously manipulate the same basic buffer contents is to copy the buffer repeatedly before sending the frame and then to continue working on the duplicate.

The buffer management code provides a variety of functions for buffer duplication services. All the functions copy the buffer header and data and realign the internal pointers to the new buffer.

Note Buffer duplication is a relatively complicated task, so it must be done with the functions provided by the buffer management code. Do not use `bcopy()` or its equivalent functions. This will result in memory corruption.

Like `getbuffer()`, the buffer duplication functions can fail. Therefore, you must always check their return values.

5.5.9.2 Duplicate a Buffer Only

The `pak_duplicate()` function duplicates the buffer, without duplicating its context and without recentering the buffer's header. A call to `pak_duplicate()` always copies all the data area from the source buffer to the duplicate.

```
paktype *pak_duplicate(paktype *pak);
```

Example: Duplicate a Buffer Only

The following example shows how to duplicate a buffer only. This example duplicates the buffer pointed to b `pak` and returns a pointer to it. The new header can then be rewritten without the risk of losing the incoming one.

```
newpak = pak_duplicate(pak);
if (newpak == NULL)
    return;
newxns = (xnshdrtype *)xnsheadstart(newpak);
newpak->if_output = pak->if_input;
newxns->cksum = 0;
```

5.5.9.3 Duplicate a Buffer and Its Context

Using `pak_duplicate()`, which always copies all the data area from the source buffer to the duplicate, can be inefficient if the contents of the original buffer are small. If the copy of the original buffer is not going to grow, it can be more efficient to use the `pak_copy()` function. This function copies the entire encapsulation area of the buffer plus `datagramsize` bytes of the network header and payload. Because `datagramsize` normally includes the encapsulation area, `pak_copy()` copies a little more data than is required. However, it can still be far more efficient than `pak_duplicate()`.

To duplicate the buffer and its context, use the `pak_copy()` function.

```
void pak_copy(paktype *source, paktype *destination, int size);
```

Example: Duplicate a Buffer and Its Context

The following example shows how to duplicate a buffer and its context. This example duplicates the buffer pointed to b `pak`. The new buffer is `newpak`..

```
newpak = getbuffer(pak->datagramsize);
if (newpak == NULL)
    return;
pak_copy(pak, newpak, pak->datagramsize);
newxns = (xnshdrtype *)xnsheadstart(newpak);
newpak->if_output = pak->if_input;
newxns->cksum = 0;
```

5.5.9.4 Duplicate and Recenter a Buffer and Its Context

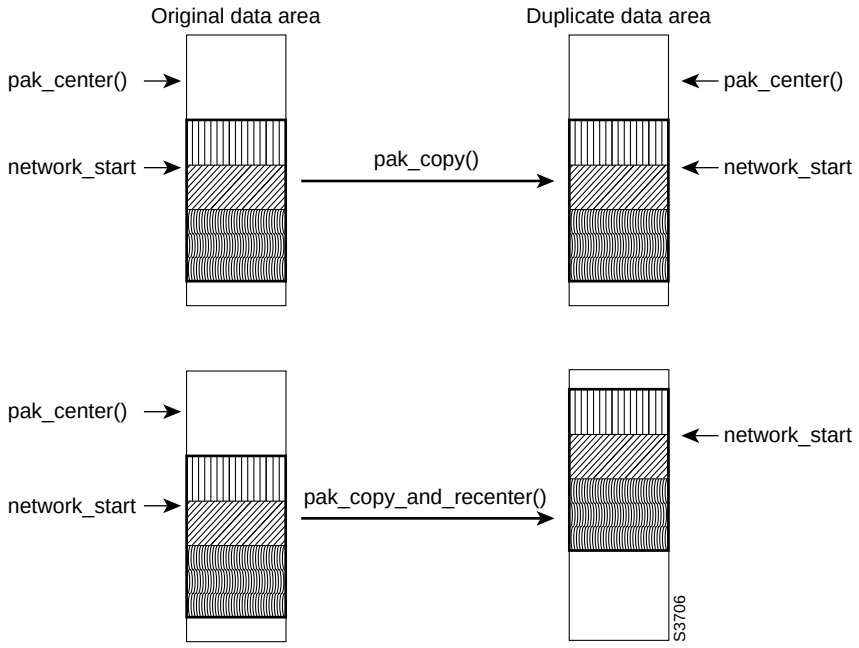
The `network_start` field in the buffer header points to the start of the network header and is usually set to be `ENCAPBYTES` from the start of the data area. However, the pointer can move, especially if there are multiple encapsulations. This can cause problems if the buffer contents are to be expanded towards the end of the data area, because the buffer might prematurely run out of space. The solution is to use the `pak_copy_and_recenter()` function to duplicate buffers. This function attempts to realign the `network_start` of the duplicated buffer to the top of the encapsulation area. This effectively snaps the buffer contents back into their optimal position.

```
void pak_copy_and_recenter(paktype *source, paktype *destination, int size);
```

5.5.9.5 Comparison of Buffer Duplication with and without Recentering

Figur e5-5 compares the `pak_copy()` and `pak_copy_and_recenter()` functions. The `pak_center` macro always points to the address at the end of the encapsulation area and is used to indicate the optimal place for `network_start`.

Figure 5-5 Comparing the pak_copy() and pak_copy_and_recenter() Functions



5.5.10 Find a Buffer Pool

To find the public buffer pool that is the best fit for a given size of buffer, use the `pak_pool_find_by_size()` function.

```
pooltype *pak_pool_find_by_size(int size);
```

5.5.11 Increase the Size of a Buffer

To increase the size of a buffer (packet), use the `pak_grow()` function.

```
paktype *pak_grow(paktype *pak, int oldsize, int newsize);
```

You must check the return value of the `pak_grow()` function, because it might allocate a new buffer and copy the packet contents into it, or in a low-memory situation it might return `NULL`. In any event, if `pak_grow()` returns non-`NULL`, the original packet pointer is invalid and should be discarded. If `pak_grow()` returns `NULL`, the original packet pointer is still valid.

5.6 Buffer Caches

5.6.1 Create a Buffer Cache

To add a buffer cache, use the `pak_pool_create_cache()` function. Adding a buffer cache to a pool initializes the structures and internal state required only; it does not add any buffers to the cache. To fill the buffer cache, use the `pool_adjust_cache()` function.

```
bool ean pak_pool_create_cache(pooltype* pool, i ntmaxsize);
```

5.6.1.1 Example: Create and Fill a Buffer Cache

The following example adds a buffer cache to `buffer_pool` that can contain a maximum of `BUFFER_POOL_CACHE_MAX` buffers. If the creation is successful, `BUFFER_POOL_CACHE_NUM` buffers are added to the cache.

```
if (pak_pool_create_cache(buffer_pool, BUFFER_POOL_CACHE_MAX) {
    pool_adjust_cache(buffer_pool, BUFFER_POOL_CACHE_NUM);
}
```

5.6.2 Remove Buffers from a Buffer Cache

When removing buffers from the buffer cache, embed the following style of code into a private buffer fetch routine. This code allows the buffer cache policy to be set on a per-user basis. In this example, the code attempts to remove a buffer from the cache using the `pool_dequeue_cache()` inline function. If this attempt fails, the code tries to obtain a buffer from the buffer pool free list. This code illustrates that the buffers that are placed into the cache list are not available to any of the `getbuffer()` calls on the pool.

```
static inline paktype *getbuf(pooltype *pool)
{
    /*
     * Attempt to get a buffer from the cache.
     */
    pak = pool_dequeue_cache(pool);

    if (!pak)
        pak = private_getbuffer(pool);

    return(pak);
}
```

5.7 Manipulate Buffers on the Input Queue of an Interface

5.7.1 Add a Buffer to the Input Queue of an Interface

To associate a buffer with an input interface, call the `set_if_input()` function. This generally needs to be done only by interface drivers that are managing a private buffer pool.

```
void set_if_input(paktype *pak, idbtype *idb);
```

To get a buffer from a public buffer pool and add it to the input queue of an interface, use the `input_getbuffer()` function. This is equivalent to calling `getbuffer()` followed by `set_if_input()`.

```
paktype *input_getbuffer(int size, idbtype *idb);
```

5.7.2 Move a Buffer to the Input Queue of Another Interface

Whenever a packet should be charged against an interface other than the one through which it arrived in the platform, it should be moved to an interface other than the input interface. Moving a buffer commonly needs to be done in tunneling code, when the headers have been removed from a tunneled packet and it needs to be reassigned from a physical interface to the software-only virtual interface for that tunnel.

To move a buffer from one input interface to another, use the `change_if_input()` function.

```
void change_if_input(paktype *pak, idbtype *newidbtype);
```

5.7.3 Remove a Buffer from the Input Queue of an Interface

When a received packet has been partially processed and should no longer be counted against the limit for an interface, it should be removed from the interface's input queue. For example, when TCP puts an out-of-sequence packet into a holding queue, it is no longer fair to charge the packet against an interface. TCP therefore calls `clear_if_input()` before saving the packet.

To remove a buffer from an input interface, call the `clear_if_input()` function.

```
void clear_if_input(paktype *pak);
```

5.8 Particles

5.8.1 Overview: Particles

The Cisco IOS software provides an extension to the general buffer scheme called *particles* that allows network frames to be constructed from several data blocks rather than as one contiguous data block. This extension allows support for scatter-gather DMA schemes within drivers, which can allow more efficient use of memory for platforms that must support interfaces with large MTUs—such as Token Ring and FDDI—with a minimal amount of buffer memory. (With scatter-gather DMA, a DMA of a contiguous block of data is spread across multiple smaller pieces of memory, for example, a 1500-byte frame is contained in three 500-byte pieces.)

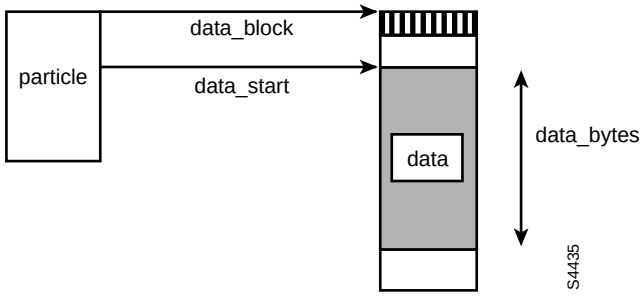
Currently, most of the Cisco IOS software expects to receive buffers that are contiguous. Therefore, all particle-based buffers sent to the process level must be coalesced into a contiguous buffer. This can impact process-level switching performance, and the use of particles within a platform's driver structure must be carefully designed and examined.

Currently, the main use for particle-based buffers in Cisco IOS software is in driver architectures that need to use small, fixed-size blocks of memory to receive into and possibly fast switch with.

5.8.2 Particle Structure

Particles consist of two fundamental blocks of memory, aparticle header and an attached data block. The header is described by the `particletype` structure. Figure 5 -6, which illustrates the structure of a particle, shows a particle with valid data of size `data_bytes` that starts at `data_start`. The `data_block` pointer points to a small information field embedded at the start of the data block that contains a magic number for block sanity checking. The usable data in the block starts immediately after this embedded header. (Some space is left before the usable data to allow room to rewrite a larger encapsulation.)

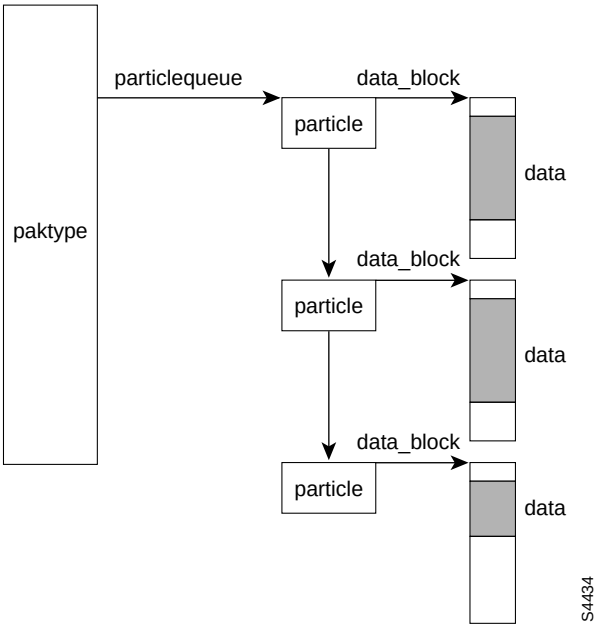
Figure 5-6 Particle Structure



When used with a buffer header, particles are chained together to form a complete frame, as illustrated in Figur e5-7. This figure shows a frame that consists of three chained particles. The shaded areas in the particle data indicate the extent of the valid frame data. Each particle header delimits the valid data in each particle data block, allowing flexible tailoring of the data considered

to be actively part of the frame. When the frame data is being parsed by the buffer support code, only the data indicated by the start and size in each particle header is considered valid. You need to make sure that these pointers are updated, especially as encapsulations change.

Figure 5-7 Chain of Particles



Note that the `paktype` header has no data block of its own, unlike prior descriptions. Creating a `paktype` pool with a size of 0 creates a pool of `paktype` headers suitable for use with particle attachments.

5.9 Particle Pools

Particles are stored in pools in much the same way that normal buffers are. The particle manager is implemented on top of the generic pool support and allows the use of all the generic pool features, such as dynamic growth and pool caches.

5.9.1 Create a Particle Pool

To create a pool to hold particles, use the `particle_pool_create()` function.

```
pooltype *particle_pool_create(char *name, int group, int size, uint flags,
                               ulong alignment, mempool *mempool);
```

5.9.2 Create a Particle Cache

To create a particle cache, use the `particle_pool_create_cache()` function. This function provides the particle-specific function vectors to a call to the `pool_cache_create()` function. Adding a particle cache to a pool initializes the structures and internal state required but does not add any particles to the cache until `pool_adjust_cache()` is called.

```
bool ean particle_pool_create_cache(pooltype* pool, i n maxsize);
```

5.9.3 Obtain a Particle from a Particle Pool

To obtain a particle from a particle pool, use the `pool_getparticle()` function. The only parameter supplied to `pool_getparticle()` is a pointer to the pool from which to obtain the particle.

```
particletype *pool_getparticle(pooltype *pool);
```

5.9.4 Return a Particle to a Pool

To return a particle to a pool, use the `retparticle()` function.

```
void retparticle(particletype *particle);
```

5.9.5 Add a Particle to the Buffer Header

To add a new particle to the end of the current list of buffer particles, use the `particle_enqueue()` function.

```
void particle_enqueue(paktype *pak, particletype *particle);
```

5.9.6 Remove a Particle from the Buffer Header

To remove a particle from the beginning of the current list of buffer particles, use the `particle_dequeue()` function.

```
particletype *particle_dequeue(paktype *pak);
```

5.9.7 Coalesce Buffers Containing Particles

Currently, only small sections of the Cisco IOS code can handle buffers consisting of particles. When a frame is passed to the process level, for example, it must be coalesced into a contiguous block of data in order for the assumptions inherent in the higher levels of code to hold true.

A buffer can be coalesced using the `pak_duplicate()` function. If a particle-based buffer is provided to the function for duplication, a contiguous buffer allocated from a public buffer pool is supplied on the return. The original buffer can then be freed and its component sections returned to their respective pools.

CHAPTER 6

Interfaces and Drivers

6.1 Interfaces: Overview

The Cisco IOS software defines an interface descriptor block (IDB) to describe the hardware and software view of an interface, and other information about the interface.

6.2 Interfaces: Historical Background

Starting with Cisco IOS Release 11.0, the implementation of IDBs changed significantly from previous releases. This section provides some history about IDBs and describes some of the factors that influenced the redesign of IDBs.

6.2.1 Growth of the IDB

Initially, the IDB contained all the information to describe the hardware configuration and application state of an interface, along with protocol-level and application-level feature variables to describe interface-specific bits of state, configuration, and status. Because of the proliferation of features, this type of implementation became unwieldy, leading to the following major problems:

- Almost all files in the Cisco IOS source code must include the `h/interface.h` header file in order to access the definition of their feature variables in the interface descriptor. Therefore, when `h/interface.h` is modified, every developer must recompile much or all their source when they incorporate updates into their development trees.
- The IDB structures have grown quite large and describe features that do not apply to every type of interface in the router. Some fields in the IDB structures might apply to none of the interfaces currently installed in the router. An example of this is the large number of variables in the `hwidb` structure that are not used on a LAN interface (Ethernet, Token Ring, or FDDI) or a serial interface that is not configured for LAPB or X.25 operation. This becomes an increasingly important issue as more features are added to the Cisco IOS software, yet the software has to fit into smaller hardware platforms, on which memory is the primary factor in the platform's cost.

6.2.2 Proliferation of Application Variables

When there were not many features in the router, adding application variables was inconvenient but not a fatal design flaw. Although everyone had to recompile their entire tree when anyone added, deleted, or changed a protocol-specific or an application-specific field in the IDB description, this was tolerable when the total number of files being compiled numbered in the hundreds and there

were fewer people per compilation server. Now that the Cisco IOS code has approximately 11,000 source files, with 30 percent of them including the header files that define the global interface structure only to allow access to their application variable fields, this design is unworkable.

In Software Release 9.21, subinterfaces for Frame Relay were added to the Cisco IOS software, impacting how all software would view IDBs. In adding Frame Relay subinterfaces, the IDB was bifurcated into two IDBs—a hardware IDB and a software IDB—and it became possible to have more than one software IDB associated with a single hardware IDB. A software IDB now contains only that information that is of interest to, or generated by, level 3 applications in the router. For instance, while serial interface statistics and state information reside in the hardware IDB, X.25 and LAPB status, statistics, and state information reside in the software IDB. New routing features should be able to support software IDBs whether the IDB is the first software IDB on a hardware interface, a tunnel interface, or a subinterface.

New features added to the router should allocate a new statically assigned subblock identifier and Modular Interface Naming and Numberingshould not add variables to the interface descriptors.

6.2.3 Proliferation of Interfaces

Initially, the maximum number of interfaces that a router might have was 24 to 30, if you filled an AGS+ with MEC boards. In this case, looping across the queue of all IDBs in the router, looking for the interface that matched the one you were looking for, was not difficult. Now, a router can contain hundreds of interfaces. Channelized interface cards can be added to a router that increase the number of interfaces by 24 at a time. In the future, the addition of a channelized T3 card could add several hundred hardware interfaces to the router with the addition of only one interface card. Clearly, with hundreds of hardware IDBs in the router and possibly many more subinterfaces, tunnel interfaces, and loopback interfaces configured on the router, looping through the queue of all IDBs in the router is not a scalable methodology

6.3 Scalability Changes

The following changes were made to increase the scalability of the IOS software:

- Subblocks and Private Lists (Releases 11.3 & 12.0)
- Maximum Interfaces Constant No Longer a Global Value (Release 11.3)
- Modular Interface Naming and Numbering (Release 12.0)
- Extensible Plugin Driver API, for 3600, 7200, and VIP Platforms. See this new chapter in the device driver manual, *Cisco IOS Device Drivers: Fundamentals of Architecture and Code.* (Release 12.0)
- Event-driven timers and queues. See “if_onesec Registry Removed” in the Scheduler chapter of this manual. (Releases 11.3 and 12.0)

6.3.1 Subblocks and Private Lists

You can use two methods to improve the scalability of features that need to access IDBs. The first method stores private IDB data fields in an area of memory known as a *subblock*. All subblocks of the same type are linked in a list. Code that needs to access this private IDB data can loop through the list of same-type subblocks instead of looping through all the IDBs in the router.

These subblock lists are maintained by the system. If IDBs are unlinked or removed from the router, the subblocks that exist on those IDBs are removed from the subblock lists. If the particular feature or protocol is removed from the configuration of the IDB, the subblock can be deleted and removed from the subblock list. Subblocks are discussed in greater detail in this chapter in "IDB Subblock."

The second method is to maintain a *private list* of only those IDBs on which your routing protocol or feature has been configured or enabled.You then loop through only those IDBs in the private list. Private lists are discussed in this chapter, in "Iterate a List of Private IDBs."

The system software provides common facilities to allow protocols and features to add and delete subblocks and to maintain their own private lists of IDBs. See also "Common Subblock Header."

6.3.2 Maximum Interfaces Constant No Longer a Global Value

The `MAX_INTERFACES` constant specifies the maximum number of interfaces that can be loaded onto a platform. Interfaces are numbered from one, so zero is invalid. `MAX_INTERFACES` used to be a global value. It has been changed to a per-platform value and is defined as follows:

```
#define MAX_INTERFACES idb_maximum_units
```

The `idb_maximum_units` variable is defined at compile time in the `os/platform.c` file. It is set to `platform_MAXINTERFACES`, which is defined in the various `machine/cisco_`*`xxx`*`.h` files. For this reason, you need to refer to the maximum interfaces constant slightly differently when defining a structure and allocating its space. For example, whereas before you referred to the constant in this way:

```
typedef struct footype {
     uint foo[MAX_INTERFACES];
} footype;
```

and in this:

```
fooptr = malloc(sizeof(footype));
```

now you need to do this instead:

```
typedef struct footype {
     uint foo[0];
} footype;
```

and this:

```
fooptr = malloc(sizeof(footype) + (sizeof(uint) * platform_MAXINTERFACES));
```

Having defined the structure and allocated the space in this manner illustrated above, you can access the maximum interfaces field as before.

Note The `foo[0]` field needs to be at the end of the structure.

If you are defining a variable that uses the maximum interfaces constant to determine the allocation at compilation time, you will now have to allocate it at runtime. For example, instead of defining the variable as before:

```
static hwidbtype *hwidblist[MAX_INTERFACES];
```

instead you need to define the variable with code such as this before you can use `hwidblist`:

```
static hwidbtype **hwidblist;
```

followed by this:

```
hwidblist = malloc(sizeof(hwidbtype *) * platform_MAXINTERFACES);
```

6.3.3 Modular Interface Naming and Numbering

One of the least modular aspects of the Cisco IOS software was the way that interfaces were named and parsed. As several different types of naming conventions were added to this code, it became less and less readable. In addition, it contained special code for different platforms among the code for all platforms.

Most of the system code used the namestrings provided by the IDB to display the IDB name. There were two times when translation between a platform-specific name and a pointer to an IDB needed to be possible: when the name was created as part of `idb_init_names()` in `if/interface.c`, and when the name was parsed. The name is parsed either when a user typed it in or when the configuration was read during `interface_action()` in `parser/parser_actions.c`.

The major differences between platforms was how the interface numbering was handled between the point that the interface name was typed, such as `ethernet` or `atm`, and the point that a VC number or subinterface is handled, which was the same on all platforms. For example, some platforms have unit-based numbering, such as `int ethernet 0`, some have strict slot-based numbering, such as `int fddi 3/0`, some have extended slot- based numbering, such as `int atm 0/2/0`), and some hav mixed slot/extended slot-based numbering.

The solution is to break out the way in which an interface number is created and parsed into a set of routines that a platform can include or create, depending on the platform.

6.3.3.1 Design

The basic design can be broken into two sections: creating the IDB names and parsing the IDB naming and numbering system.

6.3.3.2 Creating Interface Names

Creating the IDB names is rather simple to do: all interfaces currently call a routine called `idb_init_names()` to create an interface name when the interface is created.

A platform-dependent variable will be created, called `platform_create_interface_name`. This variable should be initialized at compile/link time and must be set as a subroutine that will create both a namestring and a short namestring and will link the appropriate fields in the hardware IDB and software IDB. For the hardware IDB, the fields are `hw_namestring` and `hw_short_namestring`. For the software IDB, they are `namestring` and `short_namestring`.

The `platform_create_interface_name` variable will be of this type:

```
void (*platform_create_interface_name)(idbtype *, boolean)
```

where the `idbtype *` argument contains a pointer to the software IDB to create the name from, and the Boolean argument indicates whether the interface is a physical hardware IDB type, in which case it has the complete numbering for the platform, or it is a dynamic interface, in which case it has a single number, for example loopback or dialer interfaces. The second argument should be TRUE for dynamic style interfaces.

All name routines should require that the `hwidb->name` field be set before calling the routine.

6.3.3.3 Parsing the IDB Naming and Numbering Syste

The current method of parsing the numbering system for an IDB is handled in the routine `interface_action()` in `parser_actions.c`. The basic method is to determine the interface name and translate it to an `iftype` entry, then to determine all numbering characteristics, then finally to search all the hardware IDBs for the one that matches the numbering and name.

The basic problem is that the translation from the correct numbering scheme to a hardware IDB is jumbled and unreadable.

The solution taken was to keep the existing code up to the point after the interface name (`atm`, `ethernet`, etc), then to call through a platform-dependent variable to parse the numbering. This variable, called `platform_find_interface()`, has quite a few arguments but follows a very simple process to translate from numbering to an IDB.

The `platform_find_interface()` routine will make several calls into the parser to match names, numbers, subinterface, etc. This is shown in more detail below.

The `platform_find_interface()` routine has the following form:

```
boolean (*platform_find_interface)(parseinfo *csb,
                                   iftype *ift,
                                   int *index,
                                   idbtype **return_swidb,
                                   interface_struct * const arg,
                                   char *help)
```

It will return TRUE if an interface has been found,or FALSE if it has not been found. The arguments are as follows:

`parseinfo *csb`—Contains the input characters, among other fields.

`iftype *ift`—Pointer to table that has information about the type of interface (name, if it is dynamic or not, range of numbers, etc).

`int *index` —Pointer to an index for error processing.

`idbtype **return_swidb`—Where the software IDB that matches the numbering should be returned.

`interface_struct * const arg`—Pointer to the `interface` structure in the parse chain. Indicates whether or not the IDB should be created by this command.

`char *help`—Pointer to the help string.

The general format of this routine should be thus:

1. Determine the correct numbering format (dynamic or not).
2. Use the `match_number()` and `match_char()` routines to determine platform numbering. If you do not get a number or character, return FALSE.
3. If a VC is possible, determine that. If you do not get a number or character, return FALSE.
4. Call the `match_subinterface()` routine to get a possible subinterface number (or to create a new subinterface), and to parse to the white space at the end.
5. If the `PARSE_NO_IDB` flag in `arg->flag` is set, call the `dummy_interface()` routine and return TRUE.
6. If the flag is not set, search through the hardware IDBs for a match based on the numbering read above.

7. If a hardware IDB that matches is found, call the routine `find_swidb_from_hwidb()` to determine the correct software IDB from the hardware IDB or to create a new software IDB. Return TRUE.
8. If no hardware IDB is matched, call the routine `find_swidb_from_hwidb()` to see if we should create that hardware IDB/software IDB. Return TRUE.

6.3.3.4 How to Add This for a Platform

Each platform will be required to add a `PLATFORM_NAME` field into its makefile. The `PLATFORM_NAME` field is linked into the `os_parser_lite_platform` subsystem, which becomes part of the KERNEL subsystem.

6.3.3.5 Generic Suppo t

Several generic support routines and files have been added for platforms that use unit or slotted numbering systems. These are as follows:

`if/interface.c`

The `idb_init_unit_names()` routine will create a unit-based namestring, such as `ethernet 0` or `dialer 5`. It requires the `unit` field to be placed in the `idb->unit` field. This routine can be used to create dynamic interface names, which is why it remains in `interface.c`

New files:

`if/if_name_unit.c`

The `parser_find_unit_interface()` routine can be the `platform_find_interface` vector value for unit interfaces.

`if/if_name_slot.c` - This file will work for platforms that have the format <name> <slot>/<unit> or <name> <unit> for dynamic interfaces, where the <slot>/<unit> fields should match the `hwidb->slot` and `hwidb->slotunit` fields respectively. For dynamic interfaces, <unit> should match `hwidb->unit`.

This file contains two routines: `parser_find_slotted_interface()` and `idb_init_slotted_names()`.

6.3.3.6 Platform-Specific Support

Platform-specific support has been added for the following platforms:

- 7000 and 7500: `src-rsp/if_name_rsp.c`
- C5K RSM: `src-rsp/if_name_c5rsp.c`
- ls1010: `src-4k-ls1010/if_name_ls1010.c`

This is in addition to new numbering for the popeye chassis.

6.3.3.7 Other Information

The `name_format` field in the hardware IDB has been removed.

6.3.3.8 Testing

Testing for this change is pretty simple. If the configuration can be parsed at booting of this image, the code works.

6.3.3.9 StillTo Be Done

There are two things that have not been addressed by this fix: naming controllers and removing the `idb_name_format()` routine from the system.

Controllers are used for channelized T1, E1, and T3 types of media to create the IDBs that are used within the T1 or E1.Very similiar changes could be made to the `controller_action()` and `int_cdb_name()` routines to create platform-dependent routines.

The `idb_name_format()` routine is still used in one place outside where it has been removed. The crypto routine `crypto_chassis_has_slots()` uses it to check for high-end systems. This routine should be removed and the crypto subsystems should be altered to do the correct platform modularity.

6.3.4 Extensible Plugin Driver API

This API applies to the port adapter families: 3600, 7200, and VIP platforms. It allows the drivers on an adapter to share adapter resources when carrying out their responsibilities. See the chapter on the Extensible Plugin Driver API in the 12.0 edition of the device driver manual, *Cisco IOS Device Drivers: Fundamentals of Architecture and Code*.

6.3.5 Other Scalability Changes

Several polling implementations have been transformed into event-driven mechanisms. For a summary of these changes, see the section “if_onesec Registry Removed” in the “Scheduler” chapter of this manual.

6.4 IDB Terminology

This section expands a bit upon the history of the IDB, then defines the terms related to interfaces and interface descriptor blocks (IDBs). It includes advice on which types of subblocks to use and gives some examples of subblock creation and release.

6.4.1 Hardware and Software IDBs

A hardware IDB is an interface descriptor for a hardware interface. At least one software IDB is associated with each hardware IDB when it is created. The first software IDB is created and attached to the hardware IDB when the hardware IDB is created. More software IDBs can be added to a hardware IDB when subinterfaces are added.

6.4.2 IDB Subblock

As mentioned in a previous section, historically the interface descriptor block (IDB) has been used and abused to carry all the variables bound to a specific interface. These variables included not only those that were applicable to all types of interfaces (such as the name or unit number of the interface), but also many that either were specific to one type of interface (such as TR/SRB variables, which are applicable only to Token Ring interfaces) or took up space even hen the feature was not enabled (such as variables for AppleTalk, IP, IPX, and other protocols). Using the IDB in this manner eventually caused it to grow quite large and consume too much memory.

An intermediate solution to this problem was to collect all the IDB fields that were associated with a given protocol, driver, or feature, define in a header file a structure to contain these fields, and then move them out of the IDB, leaving behind only a pointer to the new structure. The drawback to this approach was that a compile-time dependency on the IDB definition remained in the protocol-specific code. That is, the feature, protocol, or interface implementation code could not associate their structure pointers without including the definition of the IDB in the interface stream.

To solve this last problem, subblocks were created. Subblocks remove the need to `#include h/interface_private.h` to bind an application's private variables to the IDB and then to reference them later. All that is required to retrieve a subblock is the IDB pointer and a subblock identifier.

6.5 Subblock Identifier

A subblock uses a *subblock identifier* to associate subblock private memory with a particular IDB. This identifier is nothing more complicated than an unsigned integer number that is used as a key to allow fast access to the subblock at a later time.

When an application receives a packet from an interface driver or drivers, the application typically determines the input interface from `pak->if_input`. After the input IDB pointer has been obtained, the application typically references various application-specific variables associated with this interface. raditionally, these variables were simply referenced as fields in the IDB. The following example shows how these variables were referenced to increment AppleTalk errors encountered on input:

```
idb->atalk_inputerrs++;
```

The intermediate solution to this would look like the following, where all AppleTalk variables for an interface have been collected into one structure and only a typed pointer to this structure was left in the IDB:

```
idb->atalk_variables->atalk_inputerrs++;
```

The solution using subblocks looks like the following. Pointers to subblocks are not typed. That is, you cannot dereference through them as you would a structure pointer contained within another structure. To reference an application's variables associated with a particular IDB, the subblock must first be retrieved:

```
atalk_idbvars *at_idb_vars;

at_idb_vars = idb_get_swsb(idb, SWIDB_SB_ATALK);

at_idb_vars->atalk_inputerrs++;
```

6.6 Types of Subblocks

There are two types of subblocks, depending on how they are allocated:

- Preallocated
- Dynamically allocated

Pointers to *preallocated subblocks* are allocated as part of the IDB itself. They are faster to reference for reading or writing operations because accessing them requires nothing more than an array index and dereference. However, the kernel and IDB infrastructure must have a subblock identifier allocated (currently in `h/interface.h`) when the Cisco IOS kernel and infrastructure code is compiled.

Pointers to *dynamically allocated subblocks* are allocated at run-time, and the identifier you use is generated when the pointer is allocated from a monotonically increasing number space.

6.7 Which Type of Subblock to Use

When should you use one type of subblock over another? A very good question.

Preallocated subblocks require the addition of an enumerated value to `h/interface.h`, which then necessitates recompiling most of the Cisco IOS software. However, preallocated subblocks impose a lower overhead, both to get the subblock pointer from the IDB and in memory allocation.

Dynamically allocated subblocks require no changes to any file in the Cis coIOS infrastructure code—no changes to `h/interface.h`, no changes to `h/interface_private.h`. This features offers the advantage that you can write code that attaches a value to an IDB without having to recompile the infrastructure portions of the Cisco IOS code. You have to recompile only the code that lays on topof the Cisco IOS infrastructure.

6.7.1 Example: Creating a Subblock

The example in this section shows how a preallocated subblock is typically created, using AppleTalk as an example.

The `h/interface.h` contains the enumeration for the AppleTalk subblocks:

```
typedef enum {
    ...
    SWIDB_SB_APPLE,
    ...
} swidb_sb_t;
```

In `atalk/at_globals.c`, the `atalk_init_idb()` routine, which initializes the AppleTalk subblock, is something like this. This is not the real code, but merely a stripped-down version of the real code. The `atalk_init_idb()` routine is called from `atalk_init()`, which in turn is called by the registry initialization scheme.

```
void
atalk_init_idb (idbtype *idb)
{
    atalkidbtype *atalkidb;
    swidb_sb_t sbtype;

    atalkidb = malloc(sizeof(atalkidbtype));
    if (atalkidb == NULL) {
        return;
    }

    /*
     * Set up pointers back and forth.
     */
    sbtype = SWIDB_SB_APPLE;
    if (!idb_add_swsb(idb, &sbtype, atalkidb)) {
        free(atalkidb);
        return;
    }
    atalkif_init(atalkidb, TRUE, TRUE);
}
```

6.7.2 Example: Retrieving a Subblock

To access the structure you have previously bound to the IDB with the `idb_add_swdb()` routine, you retrieve the subblock. Again, the code shown in this section is simplified example code, not the real code in the system.

Typically, when a packet arrives in the system and your feature or driver thread gets control, the input interface field `if_input` is set in the packet descriptor. This is commonly coded as `pak->if_input`. Youneed only recover your subblock from the input interface found through the packet descriptor.

```
void
etalk_enqueue (paktype* pak)
{
    atalkidbtype *atidb;

    boolean valid = FALSE;

    atidb = idb_get_swsb(pak->if_input, SWIDB_SB_APPLE);

    if (atidb && atalkif_InterfaceUp(atidb)) {
        pak->transport_start = NULL;
        atalk_pak_inithdrptr(pak);
        if (atidb->atalk_enctype == ET_ETHERTALK)
            valid = etalk_validpacket(pak);
        else
            valid = atalk_validpacket(pak);
        if (valid) {
            process_enqueue_pak(atalkQ, pak);
            return;
        }
    }
    protocol_discard(pak, atalk_running);
}
```

There are two ways to retrieve a subblock pointer. In the first way, you retrieve from the IDB a pointer to the subblock using code similar to the following. This method provides no blocking. If other threads manipulate the subblock, do not store or cache this pointer in data structures with an expectation of retrieving it later.

```
subblock-ptr = idb_get_swsb(idb, subblock-identifier);
```

The second way to retrieve a subblock pointer retrieves and locks a subblock. Code in this style increments the usage value in the subblock and prevents the subblock from being removed from the IDB by another thread until the usage count is decremented.

```
subblock-ptr = idb_use_swsb(idb, subblock-identifier);
...
idb_release_swsb(idb, subblock-identifier);
```

6.7.3 Common Subblock Header

In Cisco IOS Release 11.3, a common subblock header was created to facilitate subblock management by system software. This header contains a pointer to a subblock function table structure, a pointer back to the IDB that the subblock belongs to, and link pointers.

The addition of the common subblock header allows subblock lists to be created and managed by the system software. These lists are populated by all subblocks of the same type. For example, all Ethernet subblocks are members of the Ethernet subblock list. The registration of a new subblock in an IDB will automatically add the subblock onto the list of subblocks of the same type. Either deleting a subblock or unlinking the IDB removes the subblock from the list it is on.

Also, in Cisco IOS Release 11.3, all subblocks on an IDB are members of a *linked list*.

New macros traverse all subblocks on a given hardware or software IDB.

6.7.3.1 Private IDB List

A private IDB list contains only those interfaces that have been explicitly added to the list. It is controlled by the Cisco IOS list manager list. You should only use a private IDB list if you cannot use a subblock list. See also “Comparison of Subblocks and Private IDB Lists” and “Manipulate a Private List of IDBs” in this chapter.

6.8 Subblock and VFT Support in Release 12.0

A further enhancement in Release 12.0 that further contributes to the scalability problems discussed earlier, involves placing a virtual function table (VFT) pointer in each subblock header. This addition allows subsystem functions to be invoked from common code without having to use a registry call. The aim of the VFT is to provide a well defined API between the generic IOS code and the protocol/feature code for common events, actions and requests. Prior to IOS Release 11.3, this was done through registry calls. The method of using registry calls did not scale when large numbers of interfaces were present. All entries in the registry list were invoked regardless of whether the interface was relevant to the called function. With a VFT, only the function(s) required are executed. Different VFTs exist for the SWIDB and the HWIDB.

The subblocks on the IDB are in a linked list so that an IDB's subblocks can be traversed quickly. Subblock linked lists provide an alternative to the array structure that was in use for subblocks prior to IOS Release 11.3. The main advantage of this new IDB subblock structure is to allow code to walk the list of subblocks on an IDB and call the VFT entries. A secondary advantage is that subblocks do not have to be referenced through the IDB subblock array, allowing new subblock types to be added without touching the IDB structure.

A subblock list and subblock VFT have been added to the IDB structure. The subblock list automatically provides a method of categorizing (for the purposes of iteration) the interfaces. Adding a new subblock to an IDB automatically places the subblock on the list of all subblocks of that particular type.

For example, adding a new serial interface creates a serial subblock. When that serial subblock is added to the IDB, the serial subblock is placed on a list of all serial subblocks. This provides a method of accessing via one list all serial interfaces. The use of this list avoids instances of the FOR_ALL_HWIDBS macro, helping to resolve one particular scaling problem.

With the addition of the VFT, actions that have been handled via a registry call (possible to many clients) can now be processed via the VFT functions. This can eliminate unnecessary CPU activity. To facilitate this process, the subblocks are maintained as a linked list off their IDB.

6.8.1 Implementation Details

The structure of the common HWIDB and SWIDB subblock is:

*next subblock of same type

*next subblock of this IDB

*IDB

usage count

*function table

application-specific data follows here . . .

The function table structure is:

integer type

*destroy vector (routine)

*enqueue vector (routine)

*unlink vector (routine)

*textual name

The code to add a subblock is very similar to the existing code, except the VFT pointer is used instead of the enum value of the subblock type:

```
serial_sb = malloc(sizeof(struct serialsb));
idb_add_hwsb(idb, &serial_vft, &serial_sb->sb);
```

The client subsystem code no longer has to maintain a separate IDB list of the relevant interfaces; the subblock list can be used instead. A typical code pattern in IOS is the traversal of all the IDBs, with each iteration checking for a particular feature or protocol. This can be replaced with a traversal of the subblock list:

```
FOR_ALL_HWIDBS(idb) {
if (idb->dialerdb) {
       ddb = idb->dialerdb;
       * do something... */
       }
   }
```

The above code can be replaced by:

```
FOR_ALL_HWSB(ddb, HWIDB_SB_DIALER) {
               idb = ddb->sb.idb; /* If idb is required */
               /* Do something... */
          }
```

Only the interfaces which contain the dialer subblock are traversed.

This provides a powerful and simple framework for avoiding the dreaded FOR_ALL_HWIDBS macro, and yet the client code does not have to maintain its own list of interfaces. This is a major benefit on the scaling issue.

When an event, such as a state change, occurs on the interface, pre-11.3 code typically invokes a registry list to process the event. The function reg_invoke_if_statechange_complete is called (among many others). Depending on the particular configuration of the platform, there may be up to a dozen different subsystems that have callbacks registered. Many of these callbacks are mutually exclusive, and so a typical handler checks for the relevance of the call as follows:

```
/* Frame relay callback*/
static void fr_if_statechange_complete (idbtype *idb, hwidbtype *hwidb)
{
     if (hwidb->frame_relay_stuff == NULL)
    return; /* not interested */
     /* Do something... */
}
```

In this situation, the registry granularity cannot detect that the callback is not relevant for this particular interface. The overhead of many such calls multiplied by the number of registries that are invoked causes an unnecessarily high CPU usage.

The presence of a VFT in the subblock means that this overhead can be eliminated, since actions on interfaces can now be processed by calling the VFT functions for the subblocks attached to the IDB. This limits the processing only to the functions relevant for this interface. Typically, this is done as follows:

```
FOR_ALL_SB_ON_HWIDB(sb, hwidb)
     sb->transition(sb, TRAN_STATECHANGE_COMPLETE);
```

It may be observed that this is providing an extra level of iteration instead of calling the registry list. In fact, the registry list itself is a level of iteration. So no extra levels are being executed, but the VFT case has far fewer entries.

Functions that are not required in the VFT may be filled in with return_nothing() entries.

6.8.2 Migration Path

One important advantage of the subblock/vft scheme is that it can be incrementally implemented to avoid a major change to the existing code base. The initial code to implement the scheme is minimal. Since the existing code accessing the subblocks uses the standard accessor functions, there is little change to the bulk of the code.

6.8.2.1 Migration Example

The following steps provide a typical migration path that minimizes the code disruption, allow incremental change, and support stepwise testing. Assume for the purposes of the discussion that a feature such as Frame Relay handling has been converted to use the subblock/vft scheme. The Frame Relay private data structure is attached to the HWIDB via a structure data pointer called frame_relay_stuff. No subblock exists for Frame Relay

1. Add a HWIDB subblock header (hwsb_t) to the start of the Frame Relay private data structure.
2. Create a HWIDB subblock VFT block, and populate it with return_nothing entries.
3. In the code that allocates and assigns the data structure to frame_relay_stuff, add a call to idb_add_hwsb . In the code that deallocates the structure, add a call to idb_delete_hwsb. No other code needs to change; subsystem code that references the private data structure through frame_relay_stuff does not change.
4. To use the subblock list, instead of doing FOR_ALL_HWIDBS, and checking for frame_relay_stuff, the following macro can be used:

```
#define FOR_ALL_FR_SB(sb) \
FOR_ALL_HWSB(sb, HWISB_SB_FRAMERELAY)
```

6.8.2.2 Migrating Data from IDB to Subblock

For subsystems that have fields existing in the IDB, there can be a multi-step approach to migrating the data from the IDB into the subblock:

1. Create a subblock just containing the common header and add this to the IDB. This provides subblock iteration and also the VFT. The bulk of the code can still reference the fields within the IDB, so there is little code disruption or impact.
2. Leave the VFT vacant, or migrate selected routines to the VFT as required.
3. Migrate the field to the subblock and change the code to reference the subblock fields instead of the IDB. This can be done at some point later, perhaps at the start of a release.

This gives an easy migration path and lays a foundation for future work.

6.8.2.3 Comparison of Subblocks and Private IDB Lists

As a method of traversing lists of interfaces, subblocks are to be preferred over private IDB lists. The subblock/VFT approach maintains lists of subblocks. An alternative is for a subsystem to maintain its own private list, so that the relevant interfaces are accessible by traversing the list. It is expected that the subsystem will own the list, and the scope of the list will be limited to that subsystem. Private IDB lists are designed to be holding lists for IDBs of temporary interest. They are not intended to be an interface-classing mechanism. These are the main differences between this approach and the subblock lists:

- The client subsystem must have extra code to create, add, and remove IDBs from the list. The subblock lists are automatically maintained by the standard routines that add and remove the subblock from the IDB.
- The private list elements must be allocated from the heap, so extra checking must be present to ensure the list manipulation has been successful.
- The list itself is external to the IDB, so there is no linkage from the IDB to the “owning” lists. The type of private list identifies the IDB. The IDB itself does not maintain private IDB list membership information.
- The IDB lists can be maintained independently of the subblock, whereas the subblock lists rely on the allocation of a subblock for the list pointers.
- The biggest difference is the conceptual one; private IDB lists maintain the IDB-centric nature of the software, whereas the subblock lists encourage the subsystem to take a private data viewpoint. Private IDB lists do not encourage modularity. More probably the reverse is true. Since private IDB lists maintain the centrality of the IDB as opposed to the subsystem's private data.
- There are no provisions using private IDB lists to attach some kind of semantic meaning or specific actions to the class of interfaces represented by the list. That is the list is simply a list of IDBs, and there is no way of attaching meaning to the fact that the list may be all the interfaces, for example, all Ethernet interfaces. Contrast this with the VFT attached to a subblock type, which allows methods to be called that are specific to that class of interface.

6.9 Manipulate IDBs

6.9.1 Create an IDB

Two functions are provided to create interface descriptor blocks (IDBs). The `idb_create()` function creates both a hardware IDB and the first software IDB. Additional software IDBs are created with the `idb_create_subif()` function.

```
hwidbtype *idb_create(void);

idbtype *idb_create_subif(idbtype *idb, int subidbnum);
```

The `idb_create_subif()` function is called when a subinterface is configured on an already established hardware interface.

IDBs are large data structures, and creating many of them on a router that already has limited memory can be a problem because IDBs cannot be deallocated after they are no longer needed. In the future, there will be a facility to allow the deletion of either a hardware or software interface, but for Cisco IOS Release 11.0 and earlier, creation of IDBs is a one-way operation. An IDB can be unlinked from the system data structures, but it cannot be deallocated.

When either a software or hardware IDB is created, it is assigned a monotonically increasing index or unit number. Do not change this number; it should be considered the property of the kernel. The master lists of hardware and software IDBs are kept in unit-number order for use by SNMP and other applications that need to scan all interfaces in the router in unit-number order.

6.9.2 Link an IDB

After a new hardware-software interface pair has been created by calling `idb_create()`, use the `idb_enqueue()` function to link the new hardware-software interface pair to the lists of all interfaces in the router

```
void idb_enqueue(hwidbtype *new_idb);
```

Call `idb_enqueue()` only after the IDB fields of unit number, type, encapsulation size, MTU, and major dispatch function vectors have been initialized. Once an interface pair has been passed to `idb_enqueue()`, it is available to the rest of the system.

6.9.3 Iterate over a List of IDBs

To iterate over a list of IDBs, you can (in order of preference):

- Iterate the appropriate subblock list, by calling `FOR_ALL_HWSB (hwsb,sb_type)` and `FOR_ALL_SWSB (swsb,sb_type)`.
- Iterate any private IDB list, by calling `idb_for_all_on_hwlist(type, function, *argument)` and `idb_for_all_on_swlist(type, function, *argument)`.
- Iterate the list of all interfaces in the router, using the `FOR_ALL_HWIDBS` or `FOR_ALL_SWIDBS` macro.

6.9.4 Delete an IDB

There is no straightforward way to delete an IDB from the system. To delete an IDB, it is currently best to shut down the interface and simply ignore it. To unlink a hardware-software IDB pair from the lists of all hardware and software interfaces in the router and remove any or all software subinterfaces from the software interface queue, use the `idb_unlink()` function.

```
void idb_unlink(hwidbtype *hwidb);
```

Drivers that are performing error cleanup after they have allocated a hardware-software IDB pair and before they have called `idb_enqueue()` can use the `idb_free()` function to free a hardware IDB and the first software IDB on the subinterface chain.

```
void idb_free(hwidbtype *hwidb);
```

6.10 Manipulate IDB Subblocks

6.10.1 Subblocks Types

A subblock is a private data area that is attached to an IDB and is accessed through a globally assigned enumerator identifier defined in `sys/h/subblock.h`. The subblock is accessed from the IDB using this subblock identifier. A common subblock header (`hwsb_t` or `swsb_t`) is at the start of the data area of each subblock. This header is used by the system software to manage the subblock. This header also contains various pointers that allow the subblock to be linked onto the IDB and the subblock lists.

You access the subblock using a fast array lookup method or a slower search method. The method selected depends upon the specific subblock identifier used to access the particular subblock. If an identifier is defined in `sys/h/subblock.h` as greater than the `HWIDB_SB_DYNAMIC (or`

SWIDB_SB_DYNAMIC) value, the subblock will be accessed as a search on the linked list of subblocks attached to this IDB. If it is allocated as less than the dynamic identifier limit, the faster direct array lookup is used to access the subblock.

There is a trade-off between the fast and search access. The fast access increases the size of the IDB, because an array in the IDB is used to store the subblock pointers. If the subblock is to be accessed in time-critical or interrupt handler code, it should be accessed via the fast array lookup. Otherwise, access it via the slower search method.

When adding, removing, or retrieving a subblock pointer, you must use the appropriate subblock identifier as allocated in `sys/h/subblock.h`.

A subblock can contain one or more mgd_timer structures. Because subblocks are usually dynamically allocated, you must be sure that all mgd_timer structures in a subblock are stopped before freeing the subblock. Since it is harmless to call `mgd_timer_stop()` on a timer that is already stopped, you should always call `mgd_timer_stop()` on the timer before freeing the memory area. Failure to do this can cause router crashes with `mgd_timer_set_exptime_internal()` in the backtrace.

6.10.2 Subblock Function Table

A function table pointer was introduced in Cisco IOS Release 11.3. This pointer is in the subblock header and references a table of values that include the subblock type identifier and a string identifying the subblock type.

Each subsystem that uses subblocks must declare a subblock function table for each of its subblock types. Future versions of IOS will add new entries in this function table to provide a standardized API for the subblocks.

A pointer to the function table is passed as a parameter to the routine that adds a subblock to the IDB.

6.10.3 Add an IDB Subblock

To add a subblock pointer or value to a hardware or software IDB, use the `idb_add_hwsb()` or `idb_add_swsb()` function, respectively

```
boolean idb_add_hwsb(struct hwidbtype_ *idb, hwsb_ft const *ft, hwsb_t *sb);

boolean idb_add_swsb(struct idbtype_ *idb, swsb_ft const *ft, swsb_t *sb);
```

Normally, when you work with the subblock as a private data area, you declare the common subblock header (`hwsb_t` or `swsb_t`) as the first element in the private subblock structure. When calling the add function, pass the address of this common header.

6.10.4 Return a Pointer to an IDB Subbloc

To obtain a pointer to a hardware IDB subblock if you are certain that your task will not suspend, use the `idb_get_hwsb()` or `idb_get_hwsb_inline()` function. The `idb_get_hwsb_inline()` function operates with minimal overhead. If you are accessing a subblock with `idb_get_hwsb()` or `idb_get_swsb()`, you do not need to release the subblock when you are finished.

```
void *idb_get_hwsb(struct hwidbtype_ const *idb, hwidb_sb_t type);

void *idb_get_hwsb_inline(hwidbtype const *idb, const hwidb_sb_t type);
```

The `idb_get_swsb()` and `idb_get_swsb_inline()` functions provide the equivalent functionality for software IDB subblocks.

```
void *idb_get_swsb(struct idbtype_ const *idb, swidb_sb_t type);

void *idb_get_swsb_inline(const idbtype *idb, const swidb_sb_t type);
```

To obtain a pointer to a IDB subblock if your task could suspend, possibly allowing another thread to run that might delete the subblock, use the `idb_use_hwsb()` and `idb_use_swsb()` functions.

```
void *idb_use_hwsb(struct hwidbtype_ const *idb, hwidb_sb_t type);

void *idb_use_swsb(struct idbtype_ const *idb, swidb_sb_t type);
```

You can also use the `idb_use_hwsb_inline()` and `idb_use_swsb_inline()` functions to minimize the overhead of returning the pointer. However, if you use these functions, you must `#include interface_private.h`, which will be a liability in the future.

```
void *idb_use_hwsb_inline(hwidbtype const *idb, hwidb_sb_t type);

void *idb_use_swsb_inline(idbtype const *idb, swidb_sb_t type);
```

6.10.5 Traverse a List of Subbloc

To traverse all hardware or software subblocks of a certain type, call the `FOR_ALL_HWSB` or `FOR_ALL_SWSB` macro. These macros were introduced in software Release 11.3.

```
FOR_ALL_HWSB(hwsb, sb_type)
FOR_ALL_SWSB(swsb, sb_type)
```

6.10.6 Traverse Subblocks on an IDB

To traverse all subblock on a hardware or software IDB, use the `FOR_ALL_SB_ON_HWIDB` or `FOR_ALL_SB_ON_SWIDB` macro. These macros were introduced in software Release 11.3.

```
FOR_ALL_SB_ON_HWIDB(hwidb, hwsb)
FOR_ALL_SB_ON_SWIDB(swidb, swsb)
```

6.10.7 Release an IDB Subblock

Releasing an IDB subblock decrements the usage counter that is set by `idb_use_hwsb()` or `idb_use_swsb()`.

Note If you are accessing a subblock with `idb_get_hwsb()` or `idb_get_swsb()`, you do not need to release the subblock when you are finished as you need to when you are accessing a subblock with `idb_use_hwsb()` or `idb_use_swsb()`.

You release a subblock after you return a pointer to the subblock that increments the usage counter. To release an IDB subblock, use the `idb_release_hwsb()` or `idb_release_swsb()` function.

```
boolean idb_release_hwsb(hwidbtype *hwidb, hwidb_sb_t type);

boolean idb_release_swsb(idbtype *idb, swidb_sb_t type);
```

You can also use the `idb_release_hwsb_inline()` or `idb_release_swsb_inline()` function to minimize the overhead of releasing IDB subblocks. However, if you use these functions, you must `#include interface_private.h`, which will be a liability in the future.

```
boolean idb_release_hwsb_inline(struct hwidbtype_ const *idb, hwidb_sb_t type);

boolean idb_release_swsb_inline(struct idbtype_ const *idb, swidb_sb_t type);
```

6.10.8 Delete an IDB Subblock

To delete a subblock value for a given identifier from a specified hardware interface, use either the `idb_delete_hwsb()` or `idb_delete_swsb()` function.

```
boolean idb_delete_hwsb(hwidbtype *idb, hwidb_sb_t type);

boolean idb_delete_swsb(idbtype *swidb, swidb_sb_t type);
```

Alternatively use, the `hwsb_delete` or `swsb_delete` function.

```
boolean hwsb_delete(hwsb_t *hwsb);

boolean swsb_delete(swsb_t *swsb);
```

Deleting a subblock will not free the subblock's memory, but it will unlink the subblock from the IDB and remove it from its subblock lists.

6.11 Manipulate a Private List of IDBs

Prior to Cisco IOS Release 11.0, the standard technique used by applications for operations that needed to be performed on every interface was to loop across the list of all the hardware or software interfaces in the router, check a flag field in the interface descriptor, and if the application's feature was enabled, perform the function necessary.

This technique worked adequately when there were only a few interfaces in the router. However routers can now have hundreds of hardware interfaces, which implies at least one software interface per hardware interface, and possibly many more tunnel and software subinterfaces configured on top of hundreds of hardware interfaces. Looping across every interface descriptor in a router with hundreds of interfaces to find the few interfaces that have the feature in question enabled is highly inefficient. Either a shorter list or a new way of finding all interfaces with a particular feature enabled is required.

Private lists of IDBs allow router feature code to create and maintain a list of IDBs that is a list of only those IDBs that have the feature in question enabled.

6.11.1 Create a Private List of IDBs

To create a private list of IDBs by defining a list manager header for a private list of hardware or software IDBs, use the `idb_create_list()` function. The input parameter to this function specifies the type of private IDB list to create. The output parameter is a pointer to the list type.

```
boolean idb_create_list(list_header *list_hdr, char *name);
```

6.11.2 Add an IDB to a Private List

To add an IDB pointer to a previously created private IDB list, use the `idb_add_hwidb_to_list()` or `idb_add_swidb_to_list()` function. The interface specified is inserted into the list in ascending unit-number order

```
boolean idb_add_hwidb_to_list(list_header *list_hdr, char *name);

boolean idb_add_swidb_to_list(list_header *list_hdr, char *name);
```

6.11.3 Iterate a List of Private IDBs

To iterate over a private list of IDBs, call the `FOR_ALL_HWIDBS_IN_LIST` or `FOR_ALL_SWIDBS_IN_LIST` macro. The names of these macros imply that the *idb* argument must be a hardware or software IDB pointer, respectively. However, this is not true. This is merely a macro, and it would be possible to pass a pointer to either type of IDB in the *idb* parameter of either macro and to traverse (walk) a list of those IDBs. There are two macros for clarity. Typically, application software in the router walks a list of hardware interfaces or software interfaces, but not both at the same time. The duplication of functionality in the hardware and software variant of the same macro helps to make the coder's intent more clear to the reader.

```
FOR_ALL_HWIDBS_IN_LIST(list_header *list_hdr, char *name)

FOR_ALL_SWIDBS_IN_LIST(list_header *list_hdr, char *name)
```

6.11.4 Remove an IDB from a Private List

To remove an IDB pointer from a previously created private IDB list, use the `idb_remove_from_list()` function.

```
boolean idb_remove_from_list(list_header *list_hdr, char *name);
```

6.11.5 Delete a Private List of IDBs

To destroy a previously allocated private IDB list, use the `idb_destroy_list()` function.

```
boolean idb_destroy_list(list_header *list_hdr, char *name);
```

6.12 Use IDB Helper Functions

Many functions fall into a class of functions that perform very common, but uncomplicated operations on IDBs.

6.12.1 Apply a Function over a Private IDB List

To perform a Lisp-like "apply" of a function over a private list of interfaces, use either the `idb_for_all_on_hwlist()` or `idb_for_all_on_swlist()` function. These functions walk the private IDB list, calling your function and passing a pointer to the IDB and an longword argument of your choice.

```
boolean idb_for_all_on_swlist(idblist_t type, swidbfunc_t function, v oid*argument);

boolean idb_for_all_on_hwlist(idblist_t type, hwidbfunc_t function, v oid*argument);
```

6.12.2 Test an Interface for a Property

Interfaces can have one or more properties or attributes for which you might want to test. Most of these properties are maintained in a bit field of the hardware IDB. Do not directly test bits in this field of the hardware IDB, but rather use one of the `idb_is_*()` functions.

```
boolean *idb_is_atm(const idbtype *idb);
boolean *idb_is_ethernet(const idbtype *idb);
boolean *idb_is_fddi(const idbtype *idb);
boolean *idb_is_framerelay(const idbtype *idb);
boolean *idb_is_ppp(const idbtype *idb);
boolean *idb_is_sdlc(const idbtype *idb);
boolean *idb_is_smds(const idbtype *idb);
boolean *idb_is_tokenring(const idbtype *idb);
boolean *idb_is_tokenring_like(const idbtype *idb);
boolean *idb_is_virtual(const idbtype *idb);
boolean *idb_is_x25(const idbtype *idb);
```

6.13 Encapsulate a Packet

You typically encapsulate a packet just before you transmit it.

To encapsulate a packet with a lower-level (board-level) encapsulation associated with the specified virtual circuit or address, use the `idb_board_encap()` function. This function is typically called during system initialization, during the scan for devices. It is also called when the user configures a tunnel interface, although in this case, the hardware IDB does not point to a real hardware interface.

```
boolean idb_board_encap(paktype *pak, hwidbtype *idb);
```

To encapsulate a packet with a lower-level encapsulation associated with the specified L ayer3 or Layer 2 address, use the `idb_pak_vencap()` function.

```
boolean idb_pak_vencap(paktype *pak, long address);
```

6.14 Enqueue, Dequeue, and Transmit a Packet

To enqueue a packet that will be transmitted at some later time, use the `idb_queue_for_output()` function. This function is a wrapper around the `idb->oqueue` vector, and you should use it unless you have a compelling reason to call the `idb->oqueue` vector directly.

```
void idb_queue_for_output(hwidbtype *hwidb, paktype *pak, enum HEADTAIL which);
```

To dequeue the first packet that is waiting for transmission on `idb->holdq`, use the idb_dequeue_from_output() function. This function is a wrapper around the `idb->oqueue_dequeue` vector, and you should use it unless you have a compelling reason to call `idb->oqueue_dequeue` directly.

```
paktype *idb_dequeue_from_output(hwidbtype *hwidb);
```

To start sending packets that are waiting in process memory on either the output or hold queue, use `idb_start_output()` function. This function is a wrapper around the `idb->soutput` vector, and you should use it unless there is a compelling reason to call the `idb->soutput` vector directly.

```
void idb_start_output(hwidbtype *hwidb);
```

CHAPTER 7

Platform-Specific Suppor

This chapter describes the programming interface and developer hooks for handling platform-specific initialization issues, and for obtaining platform-specific strings and values.

For a general description of system initialization, see the “System Initialization” chapter.

7.1 Platform-Specific Initialization: Overview

Although many aspects of system software support are platform independent, some features are particular to certain platforms. For example, configuring the various memory regions and pools on a hardware platform is specific to that particular platform. The initialization of various hardware devices, such as timers and interrupt controllers, is also platform specific.

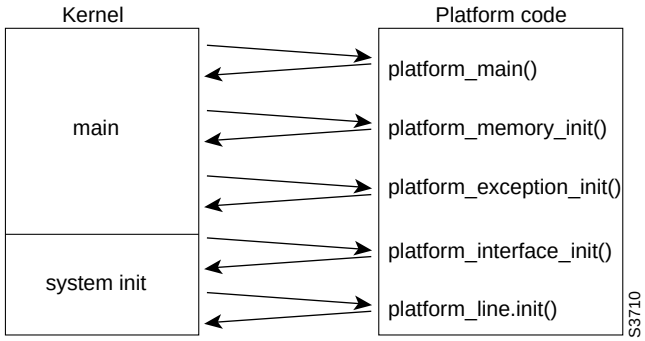
To allow platform differences to be dealt with in a nonspecific, generic way, the system code provides various general developer hooks. When the system is initialized, the system code calls these hooks to allow the platform code to initialize and configure hardware support. F igure7-1 shows the five initialization hooks that are supplied by the system code. The first three functions are called by the system code almost immediately after the system starts executing.

- `platform_main()` performs basic initialization actions, such as parsing cookie PROMs and turning on run LEDs.
- `platform_memory_init()` declares memory regions and pools to the system code using the functions supplied by the memory management code.
- `platform_exception_init()` initializes exception and interrupt handlers.

The remaining two functions are called later during initialization.

- `platform_interface_init()` initializes network interfaces.
- `platform_line_init()` initializes console and other terminal lines.

Figure 7-1 Developer Hooks for Platform Support



7.2 Fundamental Initialization

The first platform function called after the system has loaded an image and started running is `platform_main()`. This function hook allows platforms to perform absolutely basic initialization actions, such as parsing cookie PROMs and turning on run LEDs. (The cookie PROM holds all the information that is unique to a particular physical platform, such as the chassis serial number, the MAC addresses reserved for the chassis to use, the vendor [for OEM hardware], and the type of interfaces present [for nonmodular platforms].) The cookie is mapped into the platform's memory address space and is readable by the system code.

At this point in the system's initialization, no system facilities are available for use. For example, no exception handlers have been installed, nor is any memory management available. You must take great care to do the absolute minimum in this function.

When `platform_main()` is called, the only guaranteed state is that all interrupts are disabled and that BSS has been zeroed. No other state can be assumed.

7.2.1 Example: Fundamental Initialization

The following example shows how the Cisco 7000 router uses `platform_main()` to perform fundamental initialization. This hook, which is a typical application for `platform_main()`, allows the Cisco 7000 to reset outstanding interrupts left by the ROM monitor before continuing with the initialization.

```
void platform_main (void)
{
    /*
     * Reset any devices left over from ROM bugs.
     */
    reset_io_online();
}
```

7.3 Memory Initialization

One of the most critical parts of a platform's initialization is memory initialization. The function hook `platform_memory_init()` allows platforms to declare memory regions and pools to the system code using the functions supplied by the memory management code. (These functions are described in the "Memory Management" chapter.)

In a typical initialization of platform memory, the region manager declares all regions of RAM to the system code. In addition, the memory pool manager starts memory pools in the regions of memory that are to be managed as pools.

7.3.1 Example: Memory Initialization

The following example of using `platform_memory_init()` shows the memory initialization code for the Cisco 7000 platform. This code initializes the main system memory and declares the main system heap.

The first section of the memory initialization code interrogates the ROM monitor to find out how much memory is installed and declares "`main`" to be the region that describes all of system memory. The ROM monitor is responsible for sizing and initializing the main system memory in the Cisco 7000 and in most Cisco products.

```
void platform_memory_init (void)
{
    ulong memsize;

    /*
     * Find out how much main DRAM the ROM monitor thinks we have.
     */
    memsize = mon_getmemsize();

    /*
     * Add a region to describe all of main DRAM.
     */
    region_create(&main_region, "main", RAMBASE, memsize, REGION_CLASS_LOCAL,
                  REGION_FLAGS_DEFAULT);
```

The next section of the memory initialization code declares regions for each of the areas of memory related to the image. Declaring the text segment allows the system code to determine whether an instruction address is valid on certain platforms. The data segment information is used to locate and start subsystems. The heap, or processor memory pool, is usually positioned in the remaining memory for a platform. The following code declares the region from the end of BSS to the end of the main system memory to be a memory pool and starts the `MEMPOOL_CLASS_LOCAL` memory pool there.

```
    /*
     * Add regions to describe the loaded image.
     */
    region_create(&text_region, "text", TEXT_START, ((uint)TEXT_END -
(uint)TEXT_START),
                  REGION_CLASS_IMAGETEXT, REGION_FLAGS_DEFAULT);
    region_create(&data_region, "data", DATA_START,
                  ((uint)DATA_END - (uint)DATA_START), REGION_CLASS_IMAGEDATA,
                  REGION_FLAGS_DEFAULT);
    region_create(&bss_region, "bss", DATA_END, ((uint)_END - (uint)DATA_END),
                  REGION_CLASS_IMAGEBSS, REGION_FLAGS_DEFAULT);

    /*
     * Declare a region from the end of BSS to the end of main DRAM
     * and create a "local" mempool based on it.
     */
    region_create(&pmem_region, "heap", _END, (memsize -(ulong)_END),
                  REGION_CLASS_LOCAL, REGION_FLAGS_DEFAULT);
    mempool_create(&pmem_mempool, "Processor", &pmem_region, 0, NULL, 0,
                   MEMPOOL_CLASS_LOCAL);
```

The final section of the memory initialization code aliases the remaining mandatory memory pools to point at `MEMPOOL_CLASS_LOCAL`, and all memory allocations from those pool classes will be redirected to come from `MEMPOOL_CLASS_LOCAL`. These memory pools can be aliased because the Cisco 7000 has a straightforward architecture so no areas of IO or fast memory need to be declared. All mandatory memory pools must be initialized in `platform_memory_init()`. This sample routine is a basic one. Architectures with a variety of memory configurations need much more elaborate initialization code.

```
    /*
     * Add aliases for mandatory memory pools.
     */
    mempool_add_alias_pool(MEMPOOL_CLASS_IOMEM, &pmem_mempool);
    mempool_add_alias_pool(MEMPOOL_CLASS_FAST, &pmem_mempool);
    mempool_add_alias_pool(MEMPOOL_CLASS_ISTACK, &pmem_mempool);
    mempool_add_alias_pool(MEMPOOL_CLASS_PSTACK, &pmem_mempool);
}
```

7.4 Exception Initialization

The last basic initialization operations that platforms need to perform are initialization of their exception and interrupt handlers. The platform hook `platform_exception_init()` is provided for this purpose. At this point in the initialization, most platforms need to initialize their clock-tick handlers, error exception handlers, and so on. Also, the background clock, usually held in the clock variable `msclock`, must be initialized during the exception initialization.

7.4.1 Example: Exception Initialization

The following example of using `platform_exception_init()` shows the Cisco 7000 exception initialization. The call to `init_clocktick_handler()` sets up the 4-millisecond NMI clock handler and various pointers for the `msclock` support, and zeros system time. The call to `stack_hardware_init()` clears out interrupt handlers. Both these functions are provided by the 68000 CPU support in the system code.

```
void platform_exception_init (void)
{
    /*
     * Initialize the NMI handler.
     */
    init_clocktick_handler();

    /*
     * Initialize the basic exception handlers (spurious interrupts and so on).
     */
    stack_hardware_init();
}
```

7.5 Interface and Line Initialization

The final two initialization hooks—`platform_interface_init()` and `platform_line_init()`—initialize network interfaces and console lines.

Almost all network drivers in the system code are started as free-standing subsystems so that they can be easily removed from a build. However, some platforms need to perform some initialization for their network interfaces.

The goal of any `platform_interface_init()` hook should be that no references are made to actual drivers. This way, the network drivers are implemented as free-standing subsystems and are initialized via the subsystem initialization call.

You use the `platform_line_init()` hook to initialize any special TTY interfaces (such as software console support from a backplane) or make policy decisions on the number of virtual terminal (VTY) lines to be made available to the system. The console and AUX line initialization is fairly static on most Cisco platforms. Note that most parts of `platform_line_init()` are relics from earlier versions of the Cisco IOS code. This part of the code will probably undergo significant cleanups in future releases.

7.5.1 Example: Interface Initialization

The following example of using `platform_interface_init()` shows the code that the Cisco 4500 executes to initialize support for its interfaces. The initialization is straightforward.

First, the code resets all the tables that are used to bind interface structures to physical slots in the chassis. Then the ASIC that controls the network interrupts (there are two interrupt levels) is initialized. All network interfaces need a stack defined for them to use while their interrupt handlers are executed. The call to `createlevel()` creates the stack that is used on the Cisco 4500.

```
void platform_interface_init (void)
{
    int i, j;
    lev2_jumptable_t *ptr;

    /*
     * Reset the vector table to a known state.
     */
    for (i = 0; i < C4000_NUM_SLOTS; i++) {
        ptr = &lev2_jumptable[i];
        for (j = 0; j<MAX_IDB_PER_NIM; j++) {
            ptr->idb[j] = (hwidbtype *)BAD_ADDRESS;
        }
    }

    /*
     * Set the Nevada registers here.
     */
    set_nevada_regs();

    /*
     * Create the network interface interrupt stack.
     */
    createlevel(ETHER_INTLEVEL, NULL, “Network interfaces”);
```

Once the stack has been initialized, the interrupts are enabled in hardware and the high-priority handler structures are set up. Finally, service routines are registered for Cisco 4500 platform support. Although not all of these calls are strictly related to the interface support, this is a convenient place to register platform service functions.

```
        /*
         * We must now flick the switch in Nevada to enable reception of
         * high and low interrupts. Note that set_nevada_regs() has already been
         * called, so the Nevada register is configured as a control register.
         */
        IO_ASIC->sys_stat2 &= ~(SS2_LO_ENABLE|SS2_HI_ENABLE) & 0xff;

        /*
         * Create and initialize the high IRQ interrupt dispatcher.
         */
        nim_init_hi_irq_handler();

        /*
         * Add platform registry services.
         */
        reg_add_print_memory(platform_print_memory, "platform_print_memory");
        reg_add_net_cstate(nim_health_sierra_light, "nim_health_sierra_light");
        reg_add_onemin(check_for_sierra_overtemp, "check_for_sierra_overtemp");
        reg_add_subsys_init_class(nim_subsys_init_class, "nim_subsys_init_class");
    }
```

7.5.2 Example: Line Initialization

The following example of using `platform_line_init()` shows the code that the Cisco 4500 executes to initialize its console and AUX port. This code first initializes the count of console, AUX and VTY lines to be used by the platform. It then initializes the AUX and VTY lines. (The Cisco IOS system code currently assumes that there is always one console line.)

```
void platform_line_init (void)
{
    /*
     * Some default function registrations.
     */
     reg_add_tty_xon(tty_xon_default, “tty_xon_default”);

    /*
     * First discover the devices and count them.
     */
    nconlines = 1;
    nauxlines = 1;
    maxvtylines = defvtylines = nvtylines = nVTTYs;

    /*
     * Assign base indexes into the MODEMS[] data structure.
     */
    AuxBase = freeLineBase;
    freeLineBase += nauxlines;
    VTYBase = freeLineBase;

    if (protocolconversion)
        /* There are a total of 200 processes. When the system is idle,
         * 20 of them are already in use. That leaves only 180 processes
         * available for PT sessions. So, maxvtylines should not exceed 180.
         */
        maxvtylines = MAXLINES - VTYBase - MIN_NPROCS;

    ALLlines = nconlines + nttylines + nauxlines + nvtylines;

    /*
     * Initialize the MODEMS[] data structure.
     */
    auxline_init();
    vty_init();
}
```

7.6 Platform-Specific Strings

User interface commands often need to obtain platform revision, model number, and vendor derivative information. The code that handles user requests for information is common to every platform, and the `platform_get_string()` function allows you to obtain platform-specific names. This generic function allows new platforms and vendors to be easily accommodated.

To obtain platform-specific names, use the `platform_get_string()` function.

```
char *platform_get_string(platform_string_type stringtype);
```

Table 7-1Table 7-1 lists the platform strings that you can specify in the parameter that is passed to the `platform_get_string()` function. All platforms must supply the strings for `PLATFORM_STRING_NOM_DU_JOUR`, `PLATFORM_STRING_DEFAULT_HOSTNAME`, and

PLATFORM_STRING_PROCESSOR. The other strings are optional because the platform may not be able to obtain them. All the strings returned have their own storage. Because a pointer is returned to the string required, the string must not be stored on the local stack.

The platform_get_string() function returns NULL if the requested string is not implemented or is not applicable to a platform.

Table 7-1 Platform Strings

Platform String Flags	Description
PLATFORM_STRING_NOM_DU_JOUR	(Mandatory) Model number or name of the platform. Examples are “4000”and “RP1.”
PLATFORM_STRING_DEFAULT_HOSTNAME	(Mandatory) Default hostname to be used for a platform if none is specified in the configuration. This string also appears in the user interface prompt.
PLATFORM_STRING_PROCESSOR	(Mandatory) Name of the processor used by the platform. Examples are “68030,” “R4600,” and “Sparc.”
PLATFORM_STRING_VENDOR	(Optional) Vendor derivative of the platform. Examples are “Cisco,” “Cabletron,” and “Bay Networks.”
PLATFORM_STRING_PROCESSOR_REVISION	(Optional) Revision of processor used by the platform. Not all processors allow this information to be obtained.
PLATFORM_STRING_HARDWARE_REVISION	(Optional) Hardware version of the platform. This is usually the motherboard revision number. Not all platforms allow you to obtain this number.
PLATFORM_STRING_HARDWARE_SERIAL	(Optional) Serial number of the platform. This is normally held by the cookie. Not all platforms allow this to be obtained.
PLATFORM_STRING_HARDWARE_REWORK	(Optional) Hardware rework version of the actual processor board. This is effectively a subrevision of the motherboard revision number.
PLATFORM_STRING_LAST_RESET	(Optional) The reason for the last hardware reset event.

7.6.1 Examples: Platform-Specific Strings

The following is an example of calling the platform_get_string() function:

```
char *hardware_rev;

    printf(“%s %s (%s) processor”,
        platform_get_string(PLATFORM_STRING_VENDOR),
        platform_get_string(PLATFORM_STRING_NOM_DU_JOUR),
        platform_get_string(PLATFORM_STRING_PROCESSOR));

    hardware_rev = platform_get_string(PLATFORM_STRING_HARDWARE_REVISION);
    if (hardware_rev)
        printf(“ (revision %s)”, hardware_rev);
```

The preceding example produces the following output when you execute the **show version** EXEC command:

```
cisco RP1 (68040) processor
```

The following example shows the `platform_get_string()` function that is implemented on the Cisco 7000. In this example, no status can be obtained for the 68040 processor used by the RP1, nor is the hardware version or serial number available to the code. In this case, it is acceptable for `platform_get_string()` to return `NULL`.

```
char *platform_get_string (platform_string_type type)
{
    char  *value;

    switch (type) {
    case PLATFORM_STRING_NOM_DU_JOUR:
        value = “RP1”;
        break;
    case PLATFORM_STRING_DEFAULT_HOSTNAME:
        value = “Router”;
        break;
    case PLATFORM_STRING_PROCESSOR:
        value = “68040”;
        break;
    case PLATFORM_STRING_VENDOR:
        value = “cisco”;
        break;
    case PLATFORM_STRING_PROCESSOR_REVISION:
    case PLATFORM_STRING_HARDWARE_REVISION:
    case PLATFORM_STRING_HARDWARE_SERIAL:
    case PLATFORM_STRING_HARDWARE_REWORK:
    case PLATFORM_STRING_LAST_RESET:
    default:
        value = NULL;
        break;
    }
    return(value);
}
```

7.7 Platform-Specific Values

The generic system code often needs to obtain platform values that are platform specific. Rather than compile these into the code, which would prevent that code from being truly generic, the values are obtained using the `platform_get_value()` function.

```
uint platform_get_value(platform_value_type valuetype);
```

Table 7-2Table 7-2 describes the platform values that you can specify in the parameter that is passed to the `platform_get_value()` function. If a request for a value cannot obtain a value, `platform_get_value()` returns 0 by default.

Table 7-2 Platform Values

Platform Value Flags	Description
`PLATFORM_VALUE_SERVICE_CONFIG`	(Mandatory) Effectively a `boolean` that controls whether the platform should always enable the **service config** global configuration command when it boots. This command enables autoloading of configuration files from a network server. If `PLATFORM_VALUE_SERVICE_CONFIG` returns a value of 1, the system code always enables the **service config** command.
`PLATFORM_VALUE_FEATURE_SET`	(Optional) Feature set required by this platform. Use this value for feature control on platforms that use a preset cookie value to enable feature sets.

Platform Value Flags	Description
PLATFORM_VALUE_HARDWARE_REVISION	(Optional) Hardware revision of a platform. Not all platforms allow this value to be obtained.
PLATFORM_VALUE_HARDWARE_SERIAL	(Optional) Serial number of a platform. Not all platforms allow this value to be obtained.
PLATFORM_VALUE_VENDOR	(Optional) VENDOR_*xxx* value for a platform. For multivendor platforms, this is usually obtained from the cookie.
PLATFORM_VALUE_CPU_TYPE	(Mandatory) CPU_*xxx* value for a platform. This identifies the hardware class to the system code.
PLATFORM_VALUE_FAMILY_TYPE	(Mandatory) FAMILY_*xxx* value for a platform. This identifies the family class of image that the platform runs and allows incorrect images to be identified.
PLATFORM_VALUE_REFRESH_TIME	(Mandatory) Refresh time of the clock, in milliseconds. On Cisco platforms, this is usually 4 ms.
PLATFORM_VALUE_LOG_BUFFER_SIZE	(Optional) Size of the logging buffer if a platform requires that one be created by default at initialization time. If zero, no buffer is created and buffered logging is intially disabled.

7.7.1 Examples: Platform-Specific Values

The following example shows how to call the `platform_get_value()` function:

```
/*
 * Determine our processor and family type.
 */
cp   u_type= platform_get_value(PLATFORM_VALUE_CPU_TYPE);
cpu_family = platform_get_value(PLATFORM_VALUE_FAMILY_TYPE);
```

The following example shows how the system software for the Cisco 7000 dynamically obtains the CPU and family types for a platform when the platform is initializing:

```
uint platform_get_value (platform_value_type type)
{
    uint value;

    switch (type) {
    case PLATFORM_VALUE_SERVICE_CONFIG:
        value = nvsize ? FALSE : TRUE;
        break;
    case PLATFORM_VALUE_FEATURE_SET:
        value = 0x0000;
        break;
    case PLATFORM_VALUE_HARDWARE_REVISION:
        value = 0x0000;
        break;
    case PLATFORM_VALUE_HARDWARE_SERIAL:
        value = 0x0000;
        break;
    case PLATFORM_VALUE_VENDOR:
        value = VENDOR_CISCO;
        break;
    case PLATFORM_VALUE_CPU_TYPE:
        value = CPU_RP1;
        break;
    case PLATFORM_VALUE_FAMILY_TYPE:
        value = FAMILY_RP1;
        break;
    case PLATFORM_VALUE_REFRESH_TIME:
        value = REFRESHTIME;
        break;
    case PLATFORM_VALUE_LOG_BUFFER_SIZE:
        value = EIGHT_K;
        break;
    default:
        value = 0;
        break;
    }
    return(value);
}
```

CHAPTER 8

Interprocess Communications (IPC) Services

8.1 Overview: IPC Services

The Cisco IOS Interprocess Communications (IPC) services provide a communication infrastructure so that modules in a distributed system can easily interact with each other.

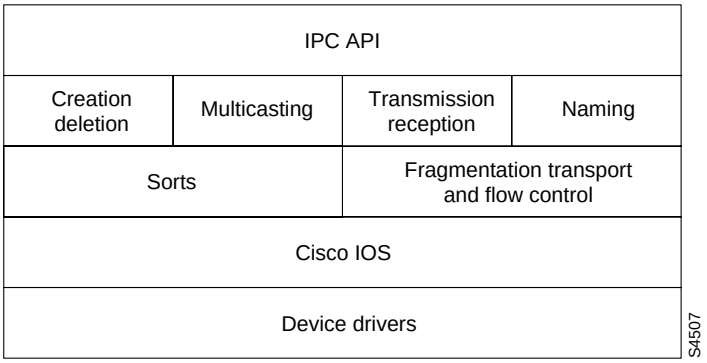
The IPC messaging system provides transparent network and local interprocess communications. To accomplish this, the following Cisco IOS IPC services are provided:

- Message transport
- Port naming and rendezvous
- Core entities, such as seat management

Multicasting will be added to the Cisco IOS IPC services in future releases of the software.

Several interfaces have been defined to provide access to IPC services. Some of these interfaces are intended for use by the upper-layer client, while others are useful at the lower layers for interfacing to other operating system or messaging system components. Figu re8-1 shows an overview of these interfaces.

Figure 8-1 IPC Message System Interfaces



8.2 Operational Environment

IPC communication services can operate in and across various environments, as listed in Table 8-1.

Table 8-1 IPC Communication Services Environments

Environment	Description
Unispace	Communication occurs on the same processor and in the same address space. Both the source and destination entities reside in a single address space.
Tightly coupled	The entities exist on different physical processors that are closely related. The processors communicate using shared memory or across a local bus with minimal delay characteristics.
Loosely coupled	The entities exist on different processors that are not closely related. The communication mechanisms differ significantly from those use for tightly coupled environments, while still being locally proximate.
Networked	The entities exist on different processors that are separated by networks. The interaction time between these entities is significantly longer than that of any of the other environments.

To achieve the flexibility to operate across a variety of environments, IPC communication provides a port with interfaces to the user, the operating system, and the underlying communication transport. These interfaces are established and controlled by the creator of the port but are hidden from the principle users of the port. Hiding the different environmental characteristics from the users of the messaging system is essential for isolating the software so that it can be migrated and distributed without requiring major changes to the software base.

User access is provided by well-defined procedure calls when the port is created. These calls provide an interface to the underlying operating system and provide the appropriate behavior. Mechanisms can include context-switchless shared memory message passing, process blocking, and message handlers providing for high-speed access and processing under special circumstances.

Communication transport access include shared memory, shared bus (such as CxBus and CyBus), raw media (such as ATM and FDDI), and network protocols (such as UDP/IP and TCP/IP).

8.3 IPC Communication: Overview

All IPC communication takes place between two cooperating entities—a source and a destination—that exchange messages. Messages are sent from source ports and received on destination ports. Source ports are specified so that receiving entities can send return messages to the originator. Ports are identified by port identifiers.

Generally, a port belongs to a single, unique entity that is responsible for processing any messages that arrive on the port. In the future, if the underlying hardware supports it, you will be able to group ports to form a multicast port so that a message can be sent to more than one destination with a single transmission.

Ports are addressed using a port identifie . One of the important characteristics of port identifiers is location independence. The sender and receiver do not have knowledge of the physical location of the destination port. Port identifiers include information about which seat (CPU) the port is homed on. Communication services are responsible for determining the destination seat and for routing the message to its destination.

In order for an entity to send or receive messages, it must create local end points, or ports, and locate destination ports.

When the entity creates a local port, it must assign a name to the port and then register the port name with the seat manager. Once the port name has been registered, the port becomes visible to potential senders.

Before an entity can send messages to a port, it must locate the destination port using the port's name.

8.4 IPC Terminology

8.4.1 Entity: Definition

An *entity* is a procedure or routine, such as a process, executing code, or a module, that makes use of IPC services. An entity uses IPC services to communicate with other cooperating entities to build distributed systems. An entity resides on a seat.

8.4.2 Message: Definition

A *message* is the basic unit of communication exchanged between entities. It includes a header, source and destination addressing information, and the message data. A message is addressed and sent to a port identifier, which identifies a port on a particular seat.

8.4.3 Port Terminology

8.4.3.1 Port

A *port* is a communications end point. There are two basic types of ports, unicast and multicast. port is identified by a 32-bit number that uniquely identifies it within the communications system. For unicast ports, the port identifier uniquely indicates both the seat and unique port within that seat.

8.4.3.2 Port Name

Each port can optionally have a *port name* associated with it. A port name is a textual name that is registered with the local seat manager and is associated with the port's identifier. A port name is unique within a seat.

8.4.3.3 Multicast Ports

Groups of ports can be aggregated and referenced as a single port so that messages can be transmitted from one source to multiple destinations. These groups of ports are called *multicast ports*. (Multicast ports are not yet supported.)

8.4.4 Port Identifier: Definition

A *port identifier* is a 32-bit integer that uniquely references a communications end point. Users of IPC services send messages to port identifiers. Each port identifier is unique.

8.4.5 Seat Terminology

8.4.5.1 Seat

A *seat* is a computational element, such as a processor, that can be communicated with using IPC services. A seat is where entities and ports reside.

8.4.5.2 Seat Manager

The *seat manager* is an entity responsible for the local seat. The seat manager is responsible for the following:

- Assigning port numbers to all its ports
- Ensuring that all local port names are unique
- Providing seat information to the zone manager
- Initializing IPC services on the seat

8.4.6 Zone Terminology

8.4.6.1 Zone

A *zone* is a collection of seats between which communications is directly possible.

8.4.6.2 Zone Manager

The *zone manager* is responsible for a group of seats that can directly communicate with each other. When a seat does not know how to communicate with another seat, it queries the zone manager. The zone manager is also responsible for resolving interseat port names.

In the simplest operational environment, IPC communication is coordinated by the seat manager and the zone manager. For other environments, such as a tightly coupled communications design (that is, shared memory), a seat can communicate directly only with other tightly coupled seats in its group.

When seats within a zone cannot communicate directly with each other, the zone manager is responsible for connecting the seats to a message-routing service.

The zone manager is also known as the IPC master

8.5 Port Naming Services

The messaging system provides only a few well-known ports. This is to encourage dynamic rendezvous and limit the amount of hard-coded information in the message system. A combination of symbolic port names and symbolic port functions is used to rendezvous with another entity or service. A distributed database is used to provide this rendezvous service. Each seat manager is responsible for maintaining the mappings between names and port identifiers for its local ports. These names can be registered when a port is created (using the `ipc_create_named_port()` function) or after a port is created (using the `ipc_register_port()` function).

Port naming services have been designed according to the following principles:

- Simple distributed database

- Both distributed and local environments
- Local function not dependent on remote services

8.5.1 Port Name Resolution

A distributed database maps symbolic names to port identifiers that are usable by the rest of the IPC communications system. Port name resolution is performed hierarchically by the following port naming components:

- Distributed name client
- Seat name server
- Zone name server

When a request is made to map a name to a port identifier, name resolution is performed hierarchically. First, the seat name server attempts to resolve local names. If the seat name server is unable to resolve the name, the request is passed to the zone name server, which requests resolution from the appropriate seat name server. This design means that the resolution of local names occurs more frequently than the resolution of intrazone names.

8.5.2 Port Name Syntax

A port name is an arbitrary string consisting of seat and function names. Each name can optionally be followed by an extension. The hierarchical naming structure of port names allows searches to take place. A port name's fully qualified name has the following syntax:

```
seat[.ext]:function[.instance]
```

The seat portion of the port name indicates where in the hierarchy the seat is located. This portion of the port name is referred to as the *server name component*. This component indicates which name servers to connect to and the level at which to allow resolution of the name server request. The function and instance portions of the name indicate a particular port on a seat and are referred to as the *function name components*.

To allow extensibility of port names, an extension can be added to each server name component. When an extension is added, port name resolution is first performed by the primary name server denoted by the base name. This allows connection to the extended service to occur. Once connection to the extended service is accomplished, name resolution proceeds from there as in a normal resolution.

A port name can take any of the following forms. Higher levels of the port name hierarchy can be specified only if the lower levels are also specified.

```
function.instance
seat[.ext]:function.instance
```

8.5.2.1 Example: Port Name Syntax

The following is an example of port name syntax. In this example, the seat name is `viper#1#6` and the port name is `fastsw.mg` .

```
viper#1#6:fastsw.mgr
```

8.5.2.2 Reserved Port Names

Table 8-2 lists the port names that are reserved for special functions.

Table 8-2 Reserved Port Names

Name	Reserved for...
IPC Master:Zone	Zone manager port for this zone.
IPC Master:Echo	Echo port for this seat.
IPC Master:Control	Control port for this seat.

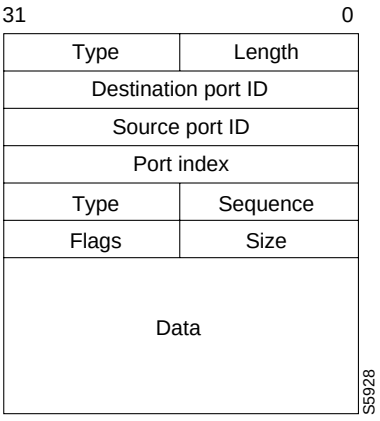
8.6 IPC Message Format

IPC messages are defined with the `ipc_message_header` structure:

```
typedef struct ipc_message_header_ {
    ipc_type_header type_hdr;
    ipc_port_id dest_port;
    ipc_port_id source_port;
    ulong port_index;
    ipc_sequence sequence;
    ipc_message_type type;
    ipc_size size;
    ipc_flags flags;
    ulong msg_id_hi;/* Timestamp*/
    ulong msg_id_lo;/* Message address */
    uchar data[0];
} ipc_message_header;
```

Figur e8-2 illustrates the IPC message format.

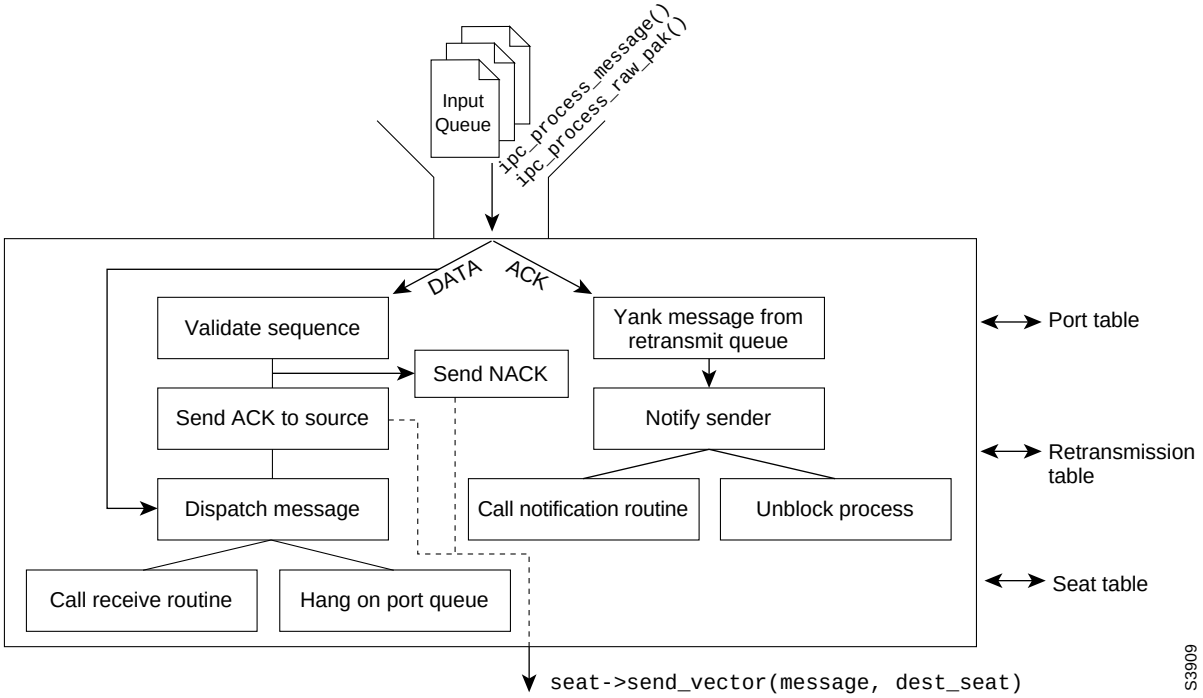
Figure 8-2 IPC Message Format



8.7 IPC Processing: Overview

Figur e8-3 illustrates basic IPC processing.

Figure 8-3 IPC Processing



8.8 Manipulate the Seat Tabl

8.8.1 Seat Table: Description

The seat table contains information about all seats in the IPC system. Entries in the table are indexed by each seat's unique 16-bit identifier.

Seat table entries contain the following information:

- Seat identifier
- Seat name
- Transport information
  - Type, such as whether the transport is local, via the ciscoBus, or via UDP
  - Output vector
  - Flags
- Protocol-specific information, such as ciscoBus slot and UDP socket information
- Information about the seat's master port.
- Packet sequencing information

An IPC seat is defined with the `ipc_seat_data_` structure:

```
struct ipc_seat_data_ {
    thread_linkage links;
    ipc_seat seat;/* The seat address */
    char *name;
    ipc_transport_type transport_type;/* Method used to get there */
    ipc_send_vector send_vector;
    boolean interrupt_ok;
    ipc_transport_t ipc_transport;
    ipc_port_info *seat_master_info;/* Info for seats master port */
    ipc_sequence last_sent;/* Last sequence number assigned */
    ipc_sequence last_heard;/* Last valid sequence number heardfrom this seat */
    ipc_sequence last_ack;/* Last ack sequence number rcv'd */
};
```

8.8.2 Create a Seat

To create a seat and assign it a name, use the `ipc_add_named_seat()` function.

```
ipc_error_code ipc_add_named_seat(uchar *seat_name, ipc_seat new_seat,
                                  ipc_transport_type transport_type,
                                  ipc_send_vector send_routine, uint transport_data
```

8.8.3 Get Information about a Seat

To retrieve information about a seat in the seat table, use the `ipc_get_seat()` function.

```
ipc_seat_data *ipc_get_seat(ipc_seat seat_address);
```

8.8.4 Reset a Seat

To reset the global IPC send and receive sequence numbers, use the `ipc_resync_seat()` function.

```
void ipc_resync_seat(ipc_seat_data *seat);
```

To reset the internal seat structures, use the `ipc_remove_seat()` function.

```
void ipc_reset_seat(ipc_seat_data *seat);
```

8.9 Manipulate the Port Table

8.9.1 Port Table: Description

The port table contains information about local ports available to users. The master server contains a complete list of ports on all servers.

Entries in the port table are indexed by a globally unique 32-bit port identifier.

Port table entries contain the following information:

- Port information
  - Port identifier
  - Port name
  - Port type

- Delivery information
  - Delivery method, for example, whether received messages should be queued or an asynchronous notification should be sent
  - Context pointer for asynchronous notification
  - Asynchronous callback vector
  - Sequencing information

An IPC port is defined with the `ipc_port_data_` structure:

```
/*
 * Port Table (Table key = port ID)
 */
struct ipc_port_data_ {
    thread_linkage links;
    ipc_port_id port;
    char *name;
    ipc_receive_method method;
    void *receive_context;
    ipc_callback receive_callback;
    watched_queue *receive_queue;
    ulong flags;
    ipc_port_seq_struct port_seq_array[IPC_PORT_MAX_OPENS];
};
```

8.9.2 Create a Port

To create a port and assign it a name, use the `ipc_create_named_port()` function.

```
ipc_error_code ipc_create_named_port(char *name, ipc_port_id *port, ulong port_flags,
                                     ipc_receive_method rx_method, void *context,
                                       void *ipc_rx_handler);
```

8.9.3 Register a Port

To register a port by its port identifier, call the `ipc_register_port()` function.

```
ipc_error_code ipc_register_port(ipc_port_id port);
```

8.9.4 Open a Port

To open a port by specifying its port identifier, use the `ipc_open_port()` function.

```
ipc_error_code ipc_open_port(ipc_port_id port_id, ipc_port_info *port_info);
```

To open a port by specifying its name, use the `ipc_open_port_by_name()` function.

```
ipc_error_code ipc_open_port_by_name(char *name, ipc_port_info *port_info);
```

An IPC port is described with the `ipc_port_info_` structure:

```
/*
 * Struct to describe port for later access by the send routine
 */
struct ipc_port_info_ {
    ipc_port_id port_id;/* IPC port id of open port */
    ulong port_features;/* What features to open port with */
    ipc_callback notify_callback;/* Callback routine for async ports */
    void *notify_context;/* Context for async ports */
    ipc_seat_data *output_seat;/* Output seat info */
    ipc_send_vector_t port_send_vector;/* IPC send vector */
    ipc_sequence last_sent;/* Last sequence number assigned */
    ipc_sequence last_heard;/* Last valid sequence number heard from this seat */
    ipc_sequence last_ack;/* Last ack sequence number rcv'd */
    ushort port_index;
};
```

Before calling these functions to open a port, the caller should initialize the following fields in the `ipc_port_info` structure:

- Port features-- Use this bitfield to set the port to IPC_PORT_FEAT_RELIABLE (reliable mode), IPC_PORT_FEAT_UNREL (unreliable mode), or IPC_PORT_FEAT_UNREL_NOT (unreliable with notification).
- Notify callback-- This field contains a function pointer to the callback routine to use with IPC_PORT_CALBACK.
- Notify context-- This field contains an optional context for the callback ack routine to use with IPC_PORT_CALLBACK.

8.9.5 Find a Por

To locate a port by name, use the `ipc_locate_port()` function.

```
ipc_port_id ipc_locate_port(ipc_name *name);
```

8.9.6 Close a Port

The close a port, use the `ipc_close_port()` function.

```
ipc_error_code ipc_close_port(ipc_port_info *port_info);
```

8.9.7 Remove a Port

To remove a port and deregister it if necessary, use the `ipc_remove_port()` function.

```
ipc_error_code ipc_remove_port(ipc_port_id port);
```

8.10 Manipulate the Message Retransmission Table

8.10.1 Message Retransmission Table: Description

The message retransmission table contains all messages sent by the local seat that have yet to be acknowledged. Entries in the table are indexed by a unique combination of the seat identifier and the packet sequence number, defined as `seat_id << 16 | sequence`. When an acknowledgment arrives, the index is calculated and the original message is retrieved quickly from the message retransmission table.

Message retransmission table entries contain the following information:

- Retransmission information
  - Sequence number
  - Retransmit timer
  - Packet pointer
- Notification
  - Callback function
  - Callback context

8.11 Send IPC Messages

8.11.1 Allocate a Message

To allocate and initialize an IPC message and its associated `paktype` data structures, use the `ipc_get_pak_message()` function.

```
ipc_message *ipc_get_pak_message(ipc_size size, ipc_port_info dest_port_info,
                                 ipc_message_type type);
```

To get a message using a user-defined buffer type, use the `ipc_get_message()` function.

```
ipc_message *ipc_get_message(ipc_size size, ipc_port_id dest_port_info,
                             ipc_message_type type, void *ipc_data,
                              void *ipc_data_buffer, ipc_free_func_t ipc_free_func);
```

8.11.2 Send a Message

To send a message, use the `ipc_send_message_blocked()` or the `ipc_send_message()` function. The first function blocks until the message is successfully sent or it times out. The second function sends the message without blocking.

```
ipc_error_code ipc_send_message_blocked(ipc_message *message,
                                        ipc_port_info *dest_port_info);

ipc_error_code ipc_send_message(ipc_message *message, ipc_port_info *dest_port_info);
```

To dispatch received packets containing IPC messages to their destination, use the `ipc_platform_init()` function.

```
void ipc_process_raw_pak(paktype *pak);
```

8.11.3 Return a Message to the IPC System

To return a message to the IPC system, use the `ipc_return_message()` function. This function returns a message to the IPC system and frees any memory buffers associated with the message.

```
void ipc_return_message(ipc_message *message);
```

8.12 Simulate RPCs

You can the IPC mechanism to implement communication between remote entities by simulating remote procedure calls (RPC).

To simulate the sending portion of a request–response operation in the IPC system, use the `ipc_send_rpc()` or `ipc_send_rpc_blocked()` function.

```
ipc_error_code ipc_send_rpc(ipc_port_info *dest_port_info, ipc_message *message);

ipc_message *ipc_send_rpc_blocked(ipc_port_info *dest_port_info,
                                  ipc_message *message, ipc_error_code *error);
```

To simulate the synchronous response portion of a request–response operation in the IPC system, use the `ipc_send_rpc_reply_blocked()` function.

```
ipc_error_code ipc_send_rpc_reply_blocked(ipc_message *original_message,
                                          ip c_message*reply_message);
```

To simulate the asynchronous response portion of a request–response operation in the IPC system, use the `ipc_send_rpc_reply()` function.

```
ipc_error_code ipc_send_rpc_reply(ipc_message *original_message,
                                  ip c_message*reply_message);
```

To set the RPC timeout period in an IPC message, use the `ipc_set_rpc_timeout()` function. The default timeout period is 1 0seconds.

```
void ipc_set_rpc_timeout (ipc_message *message, int seconds);
```

8.13 Write an IPC Application

In IPC communications, an IPC message is sent to an end point called an IPC port. In order for the port to receive messages, the port must exist and must have a callback routine associated with it.

To write an IPC application, perform the following tasks:

- Create a Port
- Open a Connection to the Port
- Send a Message

8.13.1 Create a Port

To create a port to receive IPC messages and to assign a port name to the port, use the ipc_create_named_port() function. This function returns an enumerated error code value. The following example creates a port named “Slave Registration Port”:

```
#define SIGNIN_PORT_NAME “Slave Registration Port”

ec = ipc_create_named_port(SIGNIN_PORT_NAME, &signin_port, IPC_PORT_UNICAST,
                           IPC_CALLBACK, NULL, slave_signin_handler);
if (ec != IPC_OK) {
    errmsg(&msgsym(IPC, RSP), “Master could not create named port”,
           ipc_decode_error(ec));
    return;
}
```

This code creates a port whose input is handled by the callback function `slave_signin_handler()`. The functional declaration for this callback function is as follows:

```
static void
slave_signin_handler (ipc_message *req_msg, void *context, ipc_error_code ec)
```

8.13.2 Open a Connection to the Port

Before you can open a port, you must initialize the `port_info` structure so that it requests the proper port services. The following example sets `signin_port_info.port_features` to use reliable transport mode (`IPC_PORT_FEAT_RELIABLE`) and opens a reliable-mode connection to the “Slave Registration Port”:

```
/*
 * Open the port.
 */
signin_port_info.port_features = IPC_PORT_FEAT_RELIABLE;
ipc_error = ipc_open_port_by_name(SIGNIN_PORT_NAME, &signin_port_info);

if (ipc_error != IPC_OK) {
    errmsg(&msgsym(IPC, RSP), “Slave could not find registration port”, “”);
    return;
}
```

8.13.3 Send a Message

You can send message either in blocking or nonblocking mode.

8.13.3.1 Send a Message in Blocking Mode

Send a message in blocking mode when a function must ensure that data was received by the remote IPC before proceeding. Blocking messages are similar to remote procedure calls (RPCs).

The following example gets an initialized IPC message in a `paktype` structure, copies the data into the message, and sends the message in blocking mode to a previously created test port:

```
/*
 * Transmit a message
 */
message = ipc_get_pak_message (strlen(test_message)+1, &test_port_info,
                               IPC_TYPE_SERVER_ECHO);

if (message != NULL) {
    printf(ipc_test_pass);
} else {
    printf(ipc_test_fail);
}

strcpy(message->data, test_message);

error = ipc_send_message_blocked(message, &test_port_info);

if (error == IPC_OK) {
    printf(ipc_test_pass);
} else {
    printf(ipc_test_fail_resp, ipc_decode_error(error));
}

ipc_return_message(message);
```

8.13.3.2 Send a Message in Nonblocking Mode

Send a message in nonblocking, asynchronous-style mode when the acknowledgment to the original message can be received at any time (for example, you might send nonblocking messages when sending statistics) or when sending unreliable IPC messages.

The following example of sending a message in nonblocking mode is a modified version of the blocking-mode example:

```
message = ipc_get_pak_message(strlen(test_message)+1, &test_port_info,
                              IPC_TYPE_SERVER_ECHO);

if (message != NULL) {
    printf(ipc_test_pass);
} else {
    printf(ipc_test_fail);
    return;
}

strcpy(message->data, test_message);

error = ipc_send_message(message, &test_port_info);
if (error != IPC_OK) {
    ipc_return_message(message);
}
```

8.14 Implementing IPCs on the RSP Platform

This section discusses some of the specifics for implementing IPCs on the RSP platform.

8.14.1 IPC CiscoBus Driver: Overview

The IPC core software runs on the RSP1, RSP2 (both master and slave), CIP, and VIP (all flavors) cards. The RSP1 or the master RSP2 assumes the role of the master IPC.

Each IPC card has a hardware queue associated with it. When one IPC card sends messages to another, the sending IPC card inserts the message into the receiving IPC card's hardware queue.

The queues on non-RSP IPC cards are not associated with any attention/interrupt. To receive its messages from the queue, the driver must poll its associated queue to fetch queued messages.

The queue on the RSP1/RSP2 (slave/master) card is associated with the high-level network interrupt. This interrupt is generated whenever the queue changes from empty to nonempty. Inside the interrupt handler, the RSP1/RSP2 copies the packet memory (MEMD) buffer to a DRAM buffer, then sends it to the IPC core for processing via the raw queue registration function.

A new global buffer pool is allocated for IPC message use only. The new buffer pool is managed by MEMD buffer-carving algorithm. The number of buffers is 2*n*, where *n* is the number of IPC cards that are present in the IPC system. The size of each buffe r is 4KB, which limits the size of the IPC application buffer to a maximum of 4 KB minus IPC message header overhead. Currently, there is no layer between the IPC core and CiscoBus driver that performs fragmentation and reassembly.

Some type of flow control is needed to police the buffer usage, allowing each IPC card to have its fair share of the global free MEMD buffers.

8.14.2 IPC Setup Procedure

This section describes the procedure for initializing the ciscoBus drivers on all IPC cards so that subsequent IPC operations can be performed. The procedure consists of the following phases:

- Discovery Phase
- Initialization Phase
- Registration Phase

8.14.2.1 Discovery Phase

In the discovery phase, the master IPC card discovers all the slave IPC cards that are present in the chassis. The master IPC card does this by scanning all slots, identifying the type of controller cards in the slots, and determining whether they have IPC capabilities.

The master IPC card uses the `slots[MAX_SLOTS]` structure to scan the slots. In this structure, the controller type field is `ctrlr_type`, which has an `enum` type of `ctrlr_type_t`. Currently, the following controller types support IPCs:

- `FCI_RSP1_CONTROLLER`
- `FCI_RSP2_CONTROLLER` (master and slave)
- `FCI_CIP_CONTROLLER`
- `FCI_RVIP_CONTROLLER`
- `FCI_SVIP_CONTROLLER`

The master IPC card maintains a list all discovered IPC cards in a table indexed by slot. The master IPC card stores information about the IPC cards in the `ipc_cbus_card_` and `ipc_cbus_rec_` structures.

The ipc_cbus_card_ structure contains all the information about an individual IPC card. Some of the information might be duplicated from the slots structure. You can also use slotnum to get the information indirectly from the slots structure.

```
#def       ine IPC_CARD_PRESENT 0x1/* Set if IPC card is present. */

typedef struct ipc_cbus_card_ {
    ui                  nt control;/* Control information */
    in                   t slotnum;/* Slot number of the IPC card */
    ctrl       r_type_t ctrlr_type;/* Controller type */
    in               t seat_number;/* Assigned seat number */
    rc                  v_hw_queue;/* Associated hardware queue */
    ...
} ipc_cbus_card_t;

ipc_card_t ipc_cbus_cards[MAX_SLOTS];
```

The ipc_cbus_rec_ global structure contains other IPC information:

```
typedef struct ipc_cbus_rec_ {
    bool          ean is_cbus_master;/* Set to TRUE if we are cbus master */
    boo           lean is_cbus_slave;/* Set to TRUE if we are cbus slave */
    qu              euetype messageQ;/* Queue of input messages */
...
} ipc_cbus_rec_t;

ipc_cbus_rec_t ipc_cbus_rec;
```

8.14.2.2 Initialization Phase

In the initialization phase, the master IPC card issues an initialization command to each of the discovered slave IPC cards, preparing them for normal IPC operation.

The master IPC card assigns each IPC card a unique seat number. For each slave IPC card and for the master IPC card, the seat number is the same as the slot number.

The master IPC card also assigns control port identifiers, which are used for different IPC control messages and applications. The master IPC card forms the port identifier by selecting a port number and concatenating it with the master IPC's seat number.

The master IPC card issues the ccb initialization command to a slave IPC card. This command contains the following fields:

- cmd—Command (all 16 bits)
- done—Done flag
- arg0—Slave's seat number
- res1—Master's seat number
- res0—Master's control port identifier (concatenation of the seat number and control port number)
- res1—Slave's hardware queue
- diag0—Master's hardware queue
- diag1—Unused

8.14.2.3 Registration Phase

After the slave IPC card receives an initialization command from master IPC card, the slave IPC card needs to initialize itself appropriately, then register with the master IPC card by sending an IPC control message to the master's control port.

The IPC control message that the slave IPC sends to register with the master IPC contains the following information:

- Control port number of the slave IPC
- Other information

The slave IPC control message can be extended to include other control ports, such as the echo port.

When the master IPC receives a control message from a slave IPC, the master IPC updates its ciscoBus card record list and creates a seat for the slave IPC. The master IPC then propagates the information about the slave IPC to all the other slave IPC cards by sending a control message sequentially to them. This IPC control message contains the following information, which allows all the slave IPC cards to communicate with each other directly without going through the master IPC:

- All known slaves control port identifiers
- Seat numbers of all slaves
- Hardware queues of all slaves

The master IPC ciscoBus card record list remains unchanged until the next time the master IPC card issues a command to initialize a slave IPC card.

Once the IPC slave card has registered with the master IPC card, the ciscoBus driver can support full-duplex peer-to-peer IPC communication. At this time, all IPC cards—the master and all slaves—should have the following:

- Hardware queue for receiving IPC messages
- Control port identifier for receiving IPC control messages
- List of seat numbers of each of the other IPC cards
- Hardware queue for transmitting IPC messages to each of the other IPC cards
- Control port identifier for each of the other IPC cards

8.14.3 Invoke the IPC Setup Procedure

The IPC setup procedure is invoked from the EOIR handling process. Specifically, it is invoked at the end of the EOIR handling process, after CiscoBus analysis and MEMD carving have completed.

If a non-IPC card is inserted or removed, all messages in the IPC retransmit queue are delivered to their predefined destination after EOIR handling is completed. In the worst case, there might be some delay.

If a slave IPC card is removed, the IPC messages in the retransmit queue destined for that IPC card eventually time out and are discarded. The master IPC card needs to remove from its registration table the ports that were on the removed IPC card. If any of the slave IPC cards have these ports saved in a local hash table, they need to remove them from the hash tables. Then, if an application attempts to transmit to these ports, the request is rejected and an error code is returned to the application.

If the master IPC card is removed, the slave RSP2 card comes up and assumes the role of master IPC. All control messages in the retransmit queues are discarded because they point to the removed master IPC card. All other application messages are either retransmitted or transmitted as usual.

8.14.4 Microcode Reload Handling

Microcode reloading should be transparent to the IPC core operation.

8.14.5 Implementation of the IPC CiscoBus Interface

The IPC core software on each IPC card maintains a list of its own seats and the seat on the other IPC cards.

8.14.5.1 Transmit Path

The ciscoBus driver is responsible for discovering all other IPC cards on the ciscoBus with which it needs to communicate. The driver creates seats for these IPC cards and places a transmit vector in the seat. When the IPC core software wants to transmit to a destination port identifier, the IPC software extracts the seat number, finds the corresponding seat structure, and then calls the transmit vector, passing the message and the seat structure pointer to it.

The registered ciscoBus transmit vector maps the seat to card structures that it maintains. These structures contain all ciscoBus-related transmission parameters, such as the hardware queue.

The local seat structure is an exception because it is on the same seat as the sender. This structure sends the message back to the IPC core software. This provides a communications channel between applications that are on different port numbers, but on the same seat.

In outgoing messages, the source port identifier contains the local seat number and a port number. For IPC control messages, the port number is 0. For other messages, the port number is that of the application that is sending the message. The destination port identifier of all messages contains the destination seat number and the port number of the application registered on the destination seat.

8.14.5.2 Receive Path

At the lowest layer, when the IPC hardware message queue changes from empty to nonempty, an event interrupt is generated. This interrupt invokes an event interrupt handler, which eventually calls a registry function registered from IPC ciscoBus subsystem. The registry function dequeues MEMD buffer headers (up to a certain number) from the hardware queue, copies data from MEMD buffers to system buffers, frees the MEMD buffers (headers), and places the system buffers into the `messageQ` in the IPC ciscoBus global structure `ipc_cbus_rec`.

A process created by IPC CBus subsystem initialization polls the `messageQ`. When it finds a message, the process handles IPC ciscoBus control messages locally and sends other messages to the IPC core software for processing using a registry function provided by IPC core software.

The IPC core software either processes the messages or demultiplexes them to different applications based on the destination port identifier.

8.14.6 IPC Name Service

IPC applications are aware of only the names of the remote IPC applications with which they want to communicate. These names need to map to a port identifier so that messages can actually be delivered. This mapping is provided by the *IPC name service*.

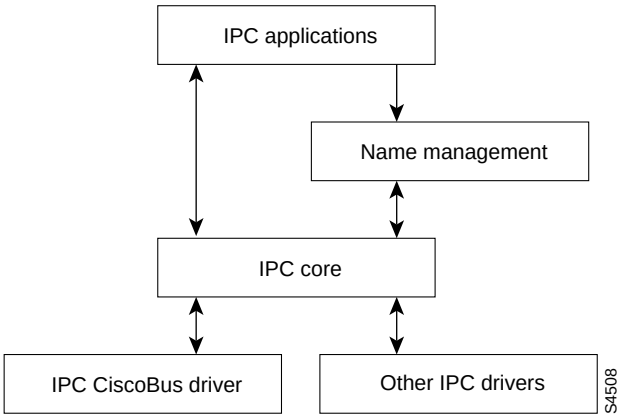
Currently, the IPC core software provides the following services:

- Name registration, which makes a newly created port known to the master IPC

- Name service, which is done by sending a query message to the master IPC and waiting for a response that contains the application's port identifier

Initially, all applications use well-known port identifiers. Thus, name registration and name service are not required. This simplifies the implementation but results in a lack of flexibility. In the next stage of the RSP IPC implementation, the application will have to register the created port and request the port identifier of remote side. Ultimately, name services will be provided to applications transparently by the IPC core software. The IPC core software will be responsible for caching and maintaining the remote name and port identifier mapping information over events such as online insertion and removal. These events also require some service from the IPC ciscoBus driver. Figur e8-4 illustrates the ultimate layered structure of these components.

Figure 8-4 IPC Application Structure



File System

9.1 Overview

The IOS File System (IFS) provides a common interface to all users of file system functionality across all platforms. This code subsumes the RSP file system and its attempt to include network devices, and extends the common POSIX-like API to all platforms and all file systems. The IFS work also creates new file systems for each data point that may be the source or destination of a file transfer (i.e downloading the AS5200 modems or dumping the SSE memory).

9.1.1 Application Level API

The application level API for all file systems is presented below.

```
extern int ifs_open(const char *path, int oflags, mode_t mode);
extern int ifs_iopen(const char *prefix, ino_t ino, int  oflags, mode_t mode);
extern int ifs_close(int fd);
extern int ifs_read(int fd, char *buf, size_t nbytes);
extern int ifs_write(int fd, char *buf, size_t nbytes);
extern int ifs_lseek(int fd, off_t offset, int whence);
extern int ifs_getdents(int fd, struct dirent *buf, size_t nbytes);
extern int ifs_chmod(const char *path, mode_t mode);
extern int ifs_rename(const char *frompath, const char *topath);
extern int ifs_remove(const char *path);
extern int ifs_mkdir(const char *path, mode_t mode);
extern int ifs_rmdir(const char *path);
extern int ifs_stat(const char *path, struct stat *stat_buf);
extern int ifs_fstat(int fd, struct stat *stat_buf);
extern int ifs_istat(const char *path, ino_t ino, struct stat *stat_buf);
extern int ifs_statfs(const char *path, struct statfs *statfs_buf);
extern int ifs_ioctl(const char *path, int function, void *arg);
extern int ifs_fioctl(int fd, int function, void *arg);
```

These routines are very close, but not always identical, to the corresponding POSIX definitions. (E.G. IFS has an ioctl function where POSIX has a devctl function.) There is also a need to provide router images running on UNIX systems, so it is convenient to have the IFS file system calls different from the UNIX file system calls.

9.1.2 Classes of File Systems

There are two classes of file systems under IFS. These are a file system containing a complete implementation of all the IFS driver vectors, and a "simple" file system that uses a framework for supporting most of the IFS functionality. The complete file system implementation should be used for any real file system (flash, disk, etc.) while the simple file system implementation is convenient for pseudo-file systems or RAM based file systems.

9.1.3 File System Types

The following general file system types are defined:

```
IFS_TYPE_FLASH
IFS_TYPE_NV
IFS_TYPE_NETWORK
IFS_TYPE_OPAQUE
IFS_TYPE_ROM
IFS_TYPE_TTY
IFS_TYPE_DISK
```

Type FLASH is for use by all file systems that use flash media. There are currently three different flash file systems in shipping product, and two more for obsolete products. Type NV is used for file system fronting NVRAM. There are currently two nvram file systems in the router. The first of these is present in all products and provides the startup configuration file and private configuration file. The second nvram file system is an overlay used on high end routers, and allows the full configuration to be saved to flash while saving a distilled configuration (no access lists) to nvram. Type NETWORK is used by all network based file systems. This currently consists of the FTP, RCP, and TFTP file systems. Type OPAQUE is used for all pseudo file systems that provide no real storage. These are often RAM based file systems such as the ATM accounting data, or interfaces to hardware devices such as the LEX card or modems. Type ROM is used by a file system on very old products that have ROM instead of bootflash. It provides access to the system images stored in the ROMs. Types TTY and DISK are for future use.

9.1.4 File System Features

The following file system feature flags are defined:

```
IFS_FEATURE_FORMAT
IFS_FEATURE_FSCK
IFS_FEATURE_VERIFY
IFS_FEATURE_UNDELETE
IFS_FEATURE_SQUEEZE
IFS_FEATURE_DIRECTORY
IFS_FEATURE_MKDIR
IFS_FEATURE_ERASE
IFS_FEATURE_RENAME
```

These feature flags simply indicate to IFS whether a file system supports a particular command, and whether this command needs to be provided to the user. If any file system in a router supports one of these commands the IFS will make the command available, but it will restrict the arguments to the command to those file systems that support it. For example if a router has a "slot0:" file system that does supports formatting and a "flash:" file system that doesn't, IFS will provide the format command to the user. If the format command was entered, the only file system argument acceptable to the command would be the "slot0:" file system.

9.1.5 File System Flags

There are many flags defined for use in describing an IFS file system. They can be found in the file sys/ifs/ifs.h. Some of the more common are: `IFS_FLAGS_ACCESS_RW` to indicate a read/write file system, `IFS_FLAGS_MEDIA_REMOTE` to indicate that the file system lives on the slave processor of a C7500 router, `IFS_FLAGS_PATH_XXX` to indicate that the file system accepts a certain component of a URL (e.g. `IFS_FLAGS_PATH_USERNAME` to indicate that a remote user name may be included), `IFS_FLAGS_LOCATION_IP` to indicate that the file system is accessed via IP, and `IFS_FLAGS_STRUCTURE_LINEAR` to indicate that a file system has a linear format and must be erased before reuse (e.g. the low end dev_io flash device driver).

9.2 Accessing File System

Accessing a file on an IFS file system is similar to accessing a file on any POSIX compliant operating system.

9.3 Implementing Simple File Systems

A simple file system provides a set of data structures describing a file (a directory, or a directory and set of files), and a set of one to four vectors for IFS to use in accessing these files. These vectors provide file system specific support for opening, closing, reading, and writing files. The framework provides generic code for these operations, plus code to handle seek, directory read, file status, file system status, and basic control functions.

9.3.1 A Trivial IFS/File System

9.3.1.1 Defining a File System

At its simplest level, the simple file system API consists of two macros, two function calls, and up to four callback routines. The API definition for this very basic file system is presented below.

The macro and routine to create a file system are:

```
#define SIFS_DECLARE_FS(identifier, mode_t mode, size_t size)
fs id _tsifs_create_fs(const char *name, ifs_type_t type, ifs_flags_t flags,
                       ifs_feature_t feature, sifs_file *root, ulong blk_size,
                       ulong blks_total, ulong blks_free);
```

The arguments to these routines are fairly self explanatory. When declaring a file system with the `SIFS_DECLARE_FS` macro, pass the name for the data structure describing the file system (this string will have "_root" appended to it), the mode of the file system using a combination of the standard definitions for user privileges (S_IRWXU, etc.), and the block size for the framework to use when collecting data being written to a file in this file system.

For example, the statement `SIFS_DECLARE_FS(acct_ready, _IRUSR | S_IXUSR, 0)` declares a C data structure named `acct_ready_root` that contains data describing a new unnamed simple file system. This file system is read-only and has a zero size.

When calling `sifs_create_fs()` to tell IFS about the file system, the first argument should be the textual prefix to use for this file system, the seconds and third argument are values from the API appropriate to this file system, the fourth argument should be a pointer to the data structure defined

by the SIFS_DECLARE_MACRO (i.e. &*name*_root), the fifth argument the block size of this device, and the last two arguments the total number of blocks on the device and the number of free blocks remaining on the device.

The file system declaration implicitly uses a pointer to the vector block for this file system. This vector block is defined below, and must be named *name*_ifs_vector.

```
typedef int (*sifs_vector_open_t)(fsid_t fsid, ifs_fdent *fdent,
                                  int oflags, mode_t mode);
typedef int (*sifs_vector_close_t)(ifs_fdent * fdent);
typedef int (*ifs_vector_read_t)(int fd, char *buf, size_t nbytes);
typedef int (*ifs_vector_write_t)(int fd, char *buf, size_t nbytes);
st ru ctsifs_vector_{
  if    s    _    v    eread;c    t    o    r    _    r    e    a    d    _
  if   s   _   v   e   write;  o   r   _   w   r   i   t   e   _
  sifs_vector_open_t   open;
  si  f  s  _  v  e  c  close; r  _  c  l  o  s  e  _
};
```

Its common for a file system to support only one or two of these vectors, and let the framework perform most of the work. The read and write routine are usually set to system provided routines for reading and writing to a memory buffer or chain of memory buffers. The file system specific work all occurs in either the open or close routines.

For example, the statement:

```
ifs_create_fs("atm-acct-ready", IFS_TYPE_OPAQUE, ACCT_IFS_FLAGS,
              AC CT_IFS_FEATURE,&acct_ready_root, 1,
             AC CT_FILEBUF_DEFAULT_SIZE,0);
```

takes the generic simple file system described by the data structure acct_ready_root, and asks IFS to install it in its tables with the name “atm-acct-ready”. This file system will be mark as opaque (i.e. other) with the specified flags and features.

9.3.1.2 Defining a File

The macro and routine to add a file to this file system are:

```
#define SIFS_DECLARE_FILE( identifier, const char *filename, ino_t inode,
                           mode_t mode)
bo olea nsifs_add_file_to_fs(const fsid_t fsid, const char *directory,
                             sifs_file *new_file);
```

The arguments to the create file macro are the name of the data structure describing the file (this string will have “_info” appended to it), the name of the file as it should appear in the file system, the inode number of the file, and the mode of the file using a combination of the standard definitions for user privileges (S_IRWXU, etc.). If desired, the file length and standard modification time values can be dynamically set in the initialization routine.

For example, the statement SIFS_DECLARE_FILE(running, “running-config”, 1, S_IRUSR | S_IWUSR) declares a C data structure named running_file that contains a description of a new simple file named “running-config”. This file will marked as inode 1 and will have read-write permissions.

When calling sifs_add_file_to_fs() to install a file in the file system, the first argument should be the file system identifier for this file system (returned by the sifs_create_fs() routine), the seconds should be the textual string of the directory where this file should be installed (usually “/”), and the last argument a pointer to the data structure created b SIFS_DECLARE_FILE (i.e. &*name*_info).

For example, the statement `sifs_add_file_to_fs(system_fsid, "/", &running_info)` takes the file described by the data structure `running_info` and asks IFS to install it in the root directory of the file system identified by the value in `system_fsid`.

9.3.1.3 Example 1 - Reading a File

Here is the complete code that implements the file system for extracting ATM statistics from a router. This example shows how to create a file system that provides read access to a data buffer in memory.

First, define flags and other information used both by this file system and by its sister file system.

```
/*
 * Common definitions
 */
#d ef     ineACCT_IFS_FLAGS(IFS_FLAGS_ACCESS_RO | IFS_FLAGS_PATH_FILENAME)
#d ef    ineACCT_IFS_FEATUREIFS_FEATURE_NONE

#d efi  neACCT_IFS_FILE_NAME"acctng_file1"
#d ef     i neACCT_FILE_INDEX1/* index for accounting code */
```

Now define the vector table that will be used for accessing this file system. Notice the null close vector indicating that there are no file system specific routines to perform on close. The read and write routines specified indicate that the file system uses a default “read from memory buffer” routine for reading, and it does not support writing to files.

```
/*
 * Forward declarations
 */
static int acct_ready_ifs_open(fsid_t fsid, ifs_fdent *fdent,
                               int oflags, mode_t mode);

/*
 * Local Storage
 */
static fsid_t acct_ready_fsid;
static sifs_vector acct_ready_ifs_vector = {
    ifs_buffer_read,
    ifs_null_write,
    acct_ready_ifs_open,
    sifs_null_close
};
```

Here is the declaration of the file system and of its single file.

```
/*
 * IFS "simple" file declarations
 */
SIFS_DECLARE_FS(acct_ready, S_IRUSR | S_IXUSR, 0);
SIFS_DECLARE_FILE(ready_file1, ACCT_IFS_FILE_NAME, 0, S_IRUSR);
```

Here is the file system specific open routine. This routine will be called after the open routine of the framework has looked up the file name, checked permissions, etc. Note that all this routine really does is set the per-file data structure to start and length of the buffer used to hold the accounting data, and sets a flag to indicate to the framework that it should not free the buffer when the file is closed. The framework will provide the data when the application reads it, handle seeks back and forth in the file, handle a stat request on the open file descriptor, and free all resources when the file is closed.

```
/*
 * acct_ready_ifs_open
 *
```

```
 * Open atm accounting ready file for accesses
 */
static int acct_ready_ifs_open (fsid_t fsid, ifs_fdent *fdent,
                                int oflags, mode_t mode)
{
    int filelen;

    /*
     * The jacket routines have checked the file permissions and guaranteed
     * that only a file read open gets to this point.  Setup the data
     * structures for receiving the data.
     */
    if (fdent->index == ready_file1_info.inode) {
        fdent->data = atmacct_getReadyFileAddrLen(ACCT_FILE_INDEX, &filelen);
        fdent->size = filelen;
        fdent->flags |= IFS_FD_FLAGS_NO_FREE;
    } else {
        /*
         * Eh?  Should be impossible
         */
        errno = ENOENT;
        return(IFS_FD_ILLEGAL);
    }
    return(fdent->fd);
}
```

This is an auxiliary routine called by the ATM accounting code when it updates the buffer containing the accounting data. This shows how to update the file size and its modification timestamp.

```
/*
 * acct_ifs_update_ready_size
 *
 * Update the file info for the ready file, and update the file system info
 * as well.
 */
void acct_ifs_update_ready_size (ulong size)
{
    ready_file1_info.size = size;
    if (clock_is_probably_valid())
        ready_file1_info.mtime = unix_time();
}
```

Here is the initialization routine for this file system. This code creates a simple IFS file system using the pre-declared file system structure, and indicates that the file system contains a total of `ACCT_FILEBUF_DEFAULT_SIZE` blocks of one byte each, and that none of the data blocks are free.

If the file system is successfully created, this routine will then set the size field in the previously declared file data structure, and add the file to the file system.

```
/*
 * acct_ready_ifs_init
 *
 * Create a file system for "atm-acct-ready"
 */
void acct_ready_ifs_init (void)
{
    int size;

    ac ct_ready_fsid=sifs_create_fs("atm-acct-ready", IFS_TYPE_OPAQUE,
                                    ACCT_IFS_FLAGS, ACCT_IFS_FEATURE,
                                    &acct_ready_root, 1,
                                    AC CT_FILEBUF_DEFAULT_SIZE,0);
    if (acct_ready_fsid == IFS_FSID_ILLEGAL)
```

```
        return;

    atmacct_getReadyFileAddrLen(ACCT_FILE_INDEX, &size);
    ready_file1_info.size = size;
    sifs_add_file_to_fs(acct_ready_fsid, "/", &ready_file1_info);
}
```

9.3.1.4 Example 2 - A More Complex Read

This example shows how to use the read routine to provide access to non-contiguous data buffers in memory. The file system is created similarly to the previous file system, but the read routine points to the following file system specific read routine.

```
/*
 * rom_ifs_read
 *
 * Read up to given number of bytes from a system device
 */
static int rom_ifs_read (int fd, char *buffer, size_t nbytes)
{
    ifs_fdent *fdent;
    int        bytes_read;

    fdent = ifs_fd_get_entry(fd);
    switch (fdent->index) {
      case ROM_IFS_IMAGE_INDEX:
        /*
         * Attempt to read count bytes from the buffer
         */
        if (((fdent->size - fdent->filepos) < nbytes) &&
            (++fdent->rom_bank < romaddr->number)) {
            bytes_read = ifs_buffer_read(fd, buffer, fdent->size);
            fdent->size = romaddr->bank[fdent->rom_bank].len;
            fdent->data = romaddr->bank[fdent->rom_bank].addr;
            fdent->filepos = 0;
            bytes_read += ifs_buffer_read(fd, buffer, nbytes - bytes_read);
            return(bytes_read);
        }
        return(ifs_buffer_read(fd, buffer, nbytes));

      default:
        /*
         * Eh? Should be impossible.
         */
        errno = EACCES;
        return(-1);
    }
}
```

This routine verifies that it is being called about the ROM Image file, providing error handling for all other files. If the read of the image file would reach the end of a bank of physical ROM and there are more physical ROMs present, the callers read request is split into two parts and the per-file data structures are updated to point to the next bank of physical ROM.

9.3.1.5 Example 3 - Writing a File

This example shows excerpts from the “system” file system for loading and parsing a new configuration file. This is a write *to* the system configuration file. A read from the system configuration file would return the current system configuration. The file system is created similarly

to the previous file systems with two exceptions. In the file system declaration a block size is provided for accumulating data from writes, and in the vector table definition a close vector is provided.

The file system declaration uses the size parameter to specify that a 16K buffer should be allocated when the configuration file is initially opened for write, and that if this buffer is filled up then additional 16K buffers will be allocated as needed until the write is complete.

```
#define SYSTEM_IFS_RUNNING_NAME  "running-config"
#define SYSTEM_IFS_RUNNING_INDEX 1
#def    ine SYSTEM_IFS_BLOCK_SIZESIXTEEN_K

SIFS_DECLARE_FILE(running, SYSTEM_IFS_RUNNING_NAME, SYSTEM_IFS_RUNNING_INDEX,
                  S_IRUSR | S_IWUSR);
SIFS_DECLARE_FS(system, S_IRUSR | S_IWUSR | S_IXUSR, SYSTEM_IFS_BLOCK_SIZE);

static void system_ifs_subsys_init (subsystype *subsys)
{
    fsid_t system_fsid;

    /*
     * Create a file system for "system"
     */
    system_root.dir_info->block_size = SYSTEM_IFS_BLOCK_SIZE;
    system_fsid = sifs_create_fs("system", IFS_TYPE_OPAQUE, SYSTEM_IFS_FLAGS,
                                 SYSTEM_IFS_FEATURE, &system_root, 0, 0, 0);
    if (system_fsid == IFS_FSID_ILLEGAL)
        return;

    /*
     * Setup the data file
     */
    sifs_add_file_to_fs(system_fsid, "/", &running_info);
}
```

The file system open routine doesn't have any work to do when opening the system "running-config" file for write. The framework sets up the data buffer that will be used, and the system provide `ifs_buffer_write()` routine will allocate additional data blocks as necessary.

```
/*
 * system_ifs_open
 *
 * Open system file for accesses
 */

static int system_ifs_open (fsid_t fsid, ifs_fdent *fdent, int oflags,
                            mode_t mode)
{
    ifs_sdent *sdent;
    ifs_sysdent *sysdent;

    /*
     * The jacket routines have checked the file permissions and guaranteed that
     * only a file read/write open gets to this point.  Setup the data structures
     * for receiving the data.
     */

    /*
     * Set some basic values
     */
    sysdent = malloc(sizeof(ifs_sysdent));
    if (!sysdent) {
        errno = ENOMEM;
        return(IFS_FD_ILLEGAL);
    }
    sdent = fdent->fd_context;
    sd e nt->fs_specific=sysdent;
    …
    …
    /*
     * Are we reading or writing this file?
     */
    if ((oflags & O_ACCMODE) != O_RDONLY) {
        /*
         * The jacket routines already took care of setting up the write data
         * buffer.  There's nothing else to do.
         */
        return(fdent->fd);
    }
    …
    … set up a configuration read here …
    …
}
```

The work of parsing the new configuration file occurs in the close routine. This routine must first concatenate together all of the data blocks accumulated as the new system file was copied, and then pass them to the system supplied `parse_configuration()` routine. This routine could (and should) be optimized to first check to see if the data is contained in a single buffer, and if so, to parse directly from that buffer and avoid the overhead of a buffer allocation and copy.

```
/*
 * system_ifs_close
 *
 * Close the opened system file
 */
static int system_ifs_close (ifs_fdent *fdent)
{
       ifs_sdent*sdent;
     ifs_sysdent*sysdent;
            char*buffer, *ptr;
             uinti, size;

    sdent = fdent->fd_context;
    sysdent = sdent->fs_specific;

    /*
     * Was this a file read or write?
     */
    if ((fdent->oflags & O_ACCMODE) == O_RDONLY) {
        /*
         * Reading.  Very little to do.  The jacket routine will take care of
         * freeing most of the memory.
         */
        free(sysdent);
        return(0);
    }

    /*
     * Writing
     */
    switch(fdent->index) {
      case SYSTEM_IFS_RUNNING_INDEX:
        /*
         * If we got an error during any running config write, this
         * bit'll be set. If it is, don't bother trying to do anything
         * with the buffer
         */
        if (fdent->flags & IFS_FD_FLAGS_ERROR)
            break;

        /*
         * Allocate a contiguous buffer to hold the text to be written
         * as a running config
         */
        buffer = malloc(fdent->size);
        if (buffer) {
            /*
             * Copy each block in turn into the contiguous buffer. The jacket routine
             * will free the individual data blocks when this routine returns.
             */
            for (i = 0, ptr = buffer; i <= fdent->block_count; i++) {
                size = (i == fdent->block_count) ?
                    fdent->block_filepos : SYSTEM_IFS_BLOCK_SIZE;
                memcpy(ptr, fdent->block[i], size);
                ptr += size;
            }
```

```
            /*
             * Parse the buffer as our new configuration
             */
            parse_configure(buffer, TRUE, sysdent->mode, sysdent->priv);
            free(buffer);
          }
          break;

        default:
          break;
      }

      free(sysdent);
      return(0);
}
```

9.3.2 Other Features

9.3.2.1 Directories

A simple file system can support the notion of sub-directories. A sub-directory is simply a file data structure with some additional information tacked onto it that will be managed by the framework. The routines for defining and subdirectory is:

```
#define SIFS_DECLARE_DIR(name, filename, mode, size)
```

The arguments to the create directory macro are the name to be used for the created data structures, the name of the directory as it should appear in the file system, the mode of the directory using a combination of the standard definitions for user privileges (S_IRWXU, etc.), and the buffer block size to use when writing to files in this directory. This routine creates two data structures. The first is the file entry that will be installed in the parent directory (this data structure name will have "_dir" appended to it). The second is the data structure used internally by the framework for maintaining a list of files in the directory.

To install a directory, use the same routine as for installing a file.

```
bo olea n s ifs_add_file_to_fs(const fsid_t fsid,const char *directory,
                               sifs_file *new_file);
```

Example 4 - Adding a directory

Here is an example of directory creation and installation from the system microcode file system.

```
SIFS_DECLARE_DIR(system_ucode, "ucode", S_IRUSR | S_IXUSR, 0);

/*
 * system_ucode_ifs_subsys_init
 *
 * system_ucode_ifs subsystem init routine.
 */
static void system_ucode_ifs_subsys_init (subsystype *subsys)
{
    fsid_t system_fsid;

    /*
     * Find the 'system' file system.
     */
    system_fsid = ifs_lookup_prefix("system:");
    if (system_fsid == IFS_FSID_ILLEGAL)
        return;

    /*
     * Create the directory for all ucode files.
     */
    if (!sifs_add_file_to_fs(system_fsid, "/", &system_ucode_dir))
        return;
    …
    … install files here …
    …
}
```

9.3.2.2 Timestamps

The simple IFS file system framework will update all file access and modification timestamps. These timestamps and the file creation timestamps can also be modified directly by the code. The file creation and update routines in Example 1 - Reading a File show a file's modification timestamp being updated.

9.4 Implementing Complete File System

A complete file system must be completely written by a developer. The complete file systems in the IOS source are (as of October 1997) are the "dev_io" file system used on the C100x, C25xx, C4x00, and C5200 platforms; the "fslib" file system used on the C7000, RSP, and LS1010 platforms; and the "malibu" flash file system used on the C3810 and LS2080 platforms.

9.4.1 IFS/File System API

Complete file systems are created by calling the ifs_create() routine. The declaration of this routine is as follows.

```
extern fsid_t ifs_create(const char *prefix, ifs_vector *vector,
                         ifs_type_t type, ifs_flags_t flags,
                         ifs_feature_t feature, void *fs_context);
```

The first argument is the URL prefix name that will be used to reference this file system. The second argument is a pointer to the vector table for this functions supported by this file system. The next three arguments describe the type, capabilities, and features of this file system. The last argument is

a magic cookie value that is defined and used only by the driver. It has no meaning to IFS, and is never referenced by IFS. Drivers can use this value to track multiple file systems that use a common set of drivers (i.e. bootflash, slot0, and slot1 on an RSP).

The vector table for a complete file system is much more complex than that for a simple file system. It includes vectors for supporting all the functions supported by the simple file system framework. The complete vector table definition is:

```
typedef int (*ifs_vector_read_t)(int fd, char *buf, size_t nbytes);
typedef int (*ifs_vector_write_t)(int fd, char *buf, size_t nbytes);
typedef int (*ifs_vector_open_t)(fsid_t fsid, const char *path, int oflags,
                                 mode_t mode);
typedef int (*ifs_vector_close_t)(int fd);
typedef int (*ifs_vector_lseek_t)(int fd, off_t offset, int whence);
typedef int (*ifs_vector_chmod_t)(fsid_t fsid, const char *path, mode_t mode);
typedef int (*ifs_vector_remove_t)(fsid_t fsid, const char *path);
typedef int (*ifs_vector_rename_t)(fsid_t fsid, const char *frompath,
                                   const char *topath);
typedef int (*ifs_vector_mkdir_t)(fsid_t fsid, const char *path, mode_t mode);
typedef int (*ifs_vector_rmdir_t)(fsid_t fsid, const char *path);
ty pe defint(*ifs_vector_getdents_t)(int fd, struct dirent *buf,
                                     size_t nbytes);
typedef int (*ifs_vector_stat_t)(fsid_t fsid, const char *path,
                                 struct stat *stat_buf);
typedef int (*ifs_vector_fstat_t)(int fd, struct stat *stat_buf);
typedef int (*ifs_vector_istat_t)(fsid_t fsid, const char *p refix,
             ino_t ino, struct stat *stat_buf);
typedef int (*ifs_vector_statfs_t)(fsid_t fsid, const char *p refix,
                                   struct statfs *statfs_buf);
typedef int (*ifs_vector_cleanup_t)(fsid_t fsid);
typedef int (*ifs_vector_ioctl_t)(fsid_t fsid, const char *path, int function,
                                  void *arg);
typedef int (*ifs_vector_fioctl_t)(int fd, int function, void *arg);
typedef int (*ifs_vector_iopen_t)(fsid_t fsid, const char *path, ino_t, int oflags,
                                  mode_t mode);


struct ifs_vector_ {
    ifs_vector_read_t     read;
    ifs_vector_write_t    write;
    ifs_vector_open_t     open;
    ifs_vector_close_t    close;
    ifs_vector_lseek_t    lseek;
    ifs_vector_chmod_t    chmod;
    ifs_vector_remove_t   remove;
    ifs_vector_rename_t   rename;
    ifs_vector_mkdir_t    mkdir;
    ifs_vector_rmdir_t    rmdir;
    ifs_vector_getdents_t getdents;
    ifs_vector_stat_t     stat;
    ifs_vector_fstat_t    fstat;
    ifs_vector_istat_t    istat;
    ifs_vector_statfs_t   statfs;
    ifs_vector_cleanup_t  cleanup;
    ifs_vector_ioctl_t    ioctl;
    ifs_vector_fioctl_t   fioctl;
    ifs_vector_iopen_t    iopen;
};
```

All of the vectors in this table are optional.

9.4.2 Common Data Structures

The following data structure is the internal representation of a file system used by IFS. It is provided here only for completeness, and should never be referenced directly. The majority of the fields are set in the call to create a file system, and there are accessor routines available for retrieving or manipulating the other field.

```
struct ifs_fsent_ {
    if      s_prefix*prefix;
    if      s_vector*vector;
    if      s_flags_tflags;
    if       s_type_ttype;
    if    s_feature_tfeature;
    in            t16num_open;
    vo            id*fs_context;
    co ns      tchar*filename_prompt;
    vo            id*show_vector;
    if  s_copy_vector*copy_vector;
};
```

The following data structure is the internal representation of a file used by IFS.

```
struct ifs_fdent_ {
    in             tfd;
    in             tnative_fd;
    fs          id_tfsid;
    ui          nt32filepos;
    ui          nt32size;
    ch           ar*data;
    ui          nt16flags;
    ui          nt16oflags;
    pi           d_tpid;
    in             tindex;
    vo           id*fd_context;

    ui            ntblock_count;
    ui            ntblock_index;
    ui            ntblock_filepos;
    ui            ntblock_size;
    ch           ar*block[IFS_MAX_BLOCKS];
};
```

The fd and fsid fields should be the file descriptor and file system identifier numbers as assigned by IFS. The oflags field should contain a copy of the flags specified in the call to the open routine, and the pid field should contain the id of the process that opened the file. The filepos and size fields are expected to maintain the current offset and total size of the file. The other fields are available for the driver to use. The native_fd and index fields are complementary; generally the native_fd field will contain the fd used by any underlying driver, or the index field will contain the file inode number used by the controlling driver. The fd_context field can be used to point to a driver specific data structure, and the flags field is used to maintain any flags that are common across multiple drivers. The data field may be used to maintain a local data buffer, and when it is the filepos field is generally an offset into this buffer. The block_xxx fields are all used by the simple file system driver described earlier.

9.4.3 Implementation

The majority of these vectors support the standard file system functions specified by the POSIX standard, and the others support standard UNIX file system functions. Only one routine is cisco specific. Any function that does not operate on an already open file descriptor will have an `i`nitial argument containing the file system identifier, allowing a single driver to support multiple file systems. After this initial file system identifier argument, these functions all take the same arguments as their POSIX or UNIX counterparts, and should perform the same functions.

The `open` routine is based on the POSIX open routine and takes four arguments. These are a file system id as determined by IFS from the path name, the path name itself (including the prefix if the file system has the `IFS_FLAGS_PATH_PREFIX` flag set), open flags from the standard set (`O_RDONL` , etc.), and the open mode. The open routine should first allocate a file descriptor using the routine `ifs_fd_create()`, and then may retrieve a pointer to the per-file data structure by using the routine `ifs_fd_get_entry()`. This file descriptor (and data structure) must be freed in the corresponding `close` routine by a call to `ifs_fd_destroy()`. At a minimum, this function should initialize the `filepos` (current position) and `size` (total size) fields in the file descriptor data structure. This function shall return the file descriptor for the file opened, or `IFS_FD_ILLEGAL` (and set `errno`) if the file is not found or an error occurred. The `iopen` vector is the equivalent of the `open` vector, only it takes an inode number as its second argument, using this instead of using a path name to specify a file.

The `close` routine is based on the POSIX close routine and takes a file descriptor as its only argument. It should perform any file system specific closing functions, and then free the system data structures with a call to `ifs_fd_destroy()`. This function shall return 0 for success, or -1 (and set `errno`) if an error occurred.

The `read` routine is based on the POSIX read routine, and takes three arguments. These are the file descriptor from which to read, a pointer to the buffer in which to place the data read, and the number of bytes to be read. This routines must update the `filepos` fields in the file descriptor data structure as it reads through the file. This routine shall return the number of bytes read, or -1 (and set `errno`) if an error occurred.

The `write` routine is based on the POSIX write routine, and also takes three arguments. These are the file descriptor to which the data shall be written, a pointer to the buffer containing the data to be written, and the number of bytes to be written. This routines must update the `filepos` and `size` fields in the file descriptor data structure as it writes to the file. This routine shall return the number of bytes written, or -1 (and set `errno`) if an error occurred.

The `lseek` routine is based on the POSIX seek routine. It takes three arguments: the file descriptor to manipulate, the number of bytes to move the file data pointer, and an enumeration indicating wither the new file position should be based on the current file position, the start of the file, or the end of the file. This routines must update the `filepos` field in the file descriptor data structure. This routine shall return the new byte offset from the beginning of the file, or -1 (and set `errno`) if an error occurred.

The `chmod` routine is based on the POSIX chmod routine, and also takes three arguments. These are a file system id as determined by IFS from the path name, the path name of the file to be changed (including the prefix if the file system has the `IFS_FLAGS_PATH_PREFIX` flag set), and the new file mode. This routine shall return the 0 for success, or -1 (and set `errno`) if an error occurred.

The `remove` routine is based on the POSIX remove routine. Its arguments are a file system id as determined by IFS from the path name, and the path name of the file to be removed (including the prefix if the file system has the `IFS_FLAGS_PATH_PREFIX` flag set). This routine shall return the 0 for success, or -1 (and set `errno`) if an error occurred.

The `rename` routine is based on the POSIX chmod routine, and also takes three arguments. These are a file system id as determined by IFS from the path name, the old path name of the file (including the prefix if the file system has the `IFS_FLAGS_PATH_PREFIX` flag set), and the new path name of the file (also including the prefix if the file system has the `IFS_FLAGS_PATH_PREFIX` flag set). This routine shall return the 0 for success, or -1 (and set `errno`) if an error occurred.

The `mkdir` routine is based on the POSIX mkdir routine, and takes three arguments. These are a file system id as determined by IFS from the path name, the path name of the new directory to be created (including the prefix if the file system has the `IFS_FLAGS_PATH_PREFIX` flag set), and the mode bits for the new directory. This routine shall return the 0 for success, or -1 (and set `errno`) if an error occurred.

The `rmdir` routine is based on the POSIX rmdir routine, and takes two arguments. These are a file system id as determined by IFS from the path name, and the path name of the directory to be destroyed (including the prefix if the file system has the `IFS_FLAGS_PATH_PREFIX` flag set). This routine shall return the 0 for success, or -1 (and set `errno`) if an error occurred.

The `getdents` routine (based on the UNI X getdents routine) is essentially a read routine for directories that is used in conjunction with the `open` and `close` routines. The POSIX equivalent is the `opendir(), readdir(), closedir()` set of functions. The getdents function takes three arguments. These are an open file descriptor for a directory entry, a pointer to a dirent data structure, and the size of the dirent data structure (i.e. a pointer to a buffer and the number of bytes to read). This routine shall return the number of bytes read, or -1 (and set `errno`) if an error occurred. ALl reads must be in multiples of the size of a dirent data structure.

The `stat` routine is based on the POSIX stat routine, and takes three arguments. These are the file system id as determined by IFS from the path name, the path name of the file for which information is desired (including the prefix if the file system has the `IFS_FLAGS_PATH_PREFIX` flag set), and a pointer to a `stat` data structure. This routine shall return 0 for success, or -1 (and set `errno`) if an error occurred. The `fstat` routine is an equivalent routine that operates on open file descriptors. It takes two arguments, an open file descriptor and a pointer to the stat buffer, and performs the same actions and returns the same values as `stat`. The istat routine is another equivalent which uses an inode number to select the file for which information is desired. Its arguments are the file system id as determined by IFS from the path name, and the prefix name of the file system (essentially redundant information), the inode number of the file for which information is desired, and a pointer to a `stat` data structure.

The `statfs` routine is based on the UNIX statfs routine, and returns information about a file system. It takes three arguments: the file system id as determined by IFS from the path name, the prefix name of the file system (essentially redundant information), and a pointer to a `statfs` data structure. This routine shall return 0 for success, or -1 (and set `errno`) if an error occurred.

The `cleanup` routine is the only function that is unique to IOS. When a file system has been removed (or is marked as such), closing the last open file on the file system will invoke the cleanup vector for that file system. This allows the file system to perform any local cleanup operations before IFS destroys all of its data structures defining the file system. If the cleanup hook returns an error code, the file system information is not destroyed.

The `ioctl` routine is based on the UNIX ioctl routine, and takes four arguments. These are the file system id as determined by IFS from the path name, the path name of the file for which information is desired (including the prefix if the file system has the `IFS_FLAGS_PATH_PREFIX` flag set), a value indicating the function to be performed, and a pointer to a function specific data structure. This routine shall return the 0 on success, or -1 (and set `errno`) if an error occurred. The `fioctl` routine is the equivalent function that operates on an open file descriptor. Instead of the first two arguments used by ioctl, it has a single argument for the file descriptor.

9.5 Additional File System Hooks

9.5.1 Copy Prompt Hook

The “copy prompt” hook allows a file system to modify the label used for the final component of a path name. This label is only used by the system “copy” command, and is used when prompting the user to input more information. The default label used is “filename”. The modem file system, for example, uses this hook to change the copy command prompt from “filename” to “modem list”. This label is more appropriate for the destination of a copy to the modem file system, since the modem file system does not emulate one file per modem, but copies any downloaded file to multiple modems. This hook should be set with the `ifs_fsid_set_filename_prompt()` routine.

9.5.2 Copy Behavior Hook

The “copy behavior” hooks allow a file system to modify the way that the copy command functions on a given file system, or to take it over entirely. There are three hooks used in different parts of the copy operation. The hook data structure is defined belo

```
ty pe defboolean(*ifs_copy_vector_setup_valid_t)(boolean prompt);
ty pe defboolean(*ifs_copy_vector_check_args_t)(void /*ifs_pathent*/ *src_pathent,
                 void /*ifs_pathent*/ *dst_pathent);
ty pe     defint(*ifs_copy_vector_copy_t)(void /*ifs_pathent*/ *src_pathent,
                 void /*ifs_pathent*/ *dst_pathent, boolean erase,
                 boolean verbose);

typedef struct ifs_copy_vector_ {
    i fs_copy_vector_setup_valid_tcheck_setup;
    i  fs_copy_vector_check_args_tcheck_args;
           ifs_copy_vector_copy_tcopy;
} ifs_copy_vector;
```

These hooks should be set with the `ifs_fsid_set_copy_vector()` routine.

The `check_setup` hook should perform any validation necessary to determine if the copy hooks have all necessary resources available to function. It takes a single argument, a boolean indicating whether or not the user can be prompted for information. If `TRUE`, the user may be prompted; if `FALSE` the copy command was generated programmatically and there is no user present to respond to a query. If this routine returns FALSE, the copy command is aborted. The `check_args` command is called after both source and destination filenames have been fully parsed (and the user prompted to enter missing pieces), and have been checked to insure that they are different files. The `check_args` routine is passed pointers to the pathent data structures describing both source and destination file names.If this routine returnsFALSE, the copy command is also aborted. The final vector, `cop` , is called to perform the actual data copy instead of the system supplied default routine. It is passed pointers to both the source and destination pathent data structures, a boolean indicating whether the destination file system should be erased, and a boolean indication whether it may print any output. This routine, if it returns, must return the number of bytes transferred. Any of these hooks may be null if the corresponding access to the copy command isn’t needed.

The best example of the use of these hooks is the “flash load helper” support used on some of the low end router products. When the router is executing directly from flash memory,the flash is configured in read-only mode and cannot be modified. The “flash load helper” file system applies a set of copy hooks to the flash file system, so that any attempts to write to flash are handled properly. The check_setup hook is used to verify that flash load helper support is present in the router. The check_args hook is used to validate that the source path uses a protocol that is known to the particular

version of flash load helper available on that system. The copy hook is used to reformat the arguments into the form used by the flash load helper code, reboot the router, and invoke the bootstrap image or rom image's copy routine.

9.5.3 "Show Flash" Hook

The "show flash" hooks are used to provide a common API for accessing data about (not on) a flash file system. These hooks are used to implement, obviously, the "show flash" command.

```
typedef void (*ifs_show_flash_all_t)(const char *prefix);
typedef void (*ifs_show_flash_chips_t)(const char *prefix);
typedef void (*ifs_show_flash_default)(const char *prefix);
typedef void (*ifs_show_flash_detailed_t)(const char *prefix);
typedef void (*ifs_show_flash_err_t)(const char *prefix);
typedef void (*ifs_show_flash_filesys_t)(const char *prefix);
typedef void (*ifs_show_flash_summary_t)(const char *prefix);

typedef struct ifs_show_flash_vector_ {
    void             ifs_show_flash_all_tshow_flash_all;
    void         ifs_show_flash_chips_tshow_flash_chips;
    void     ifs_show_flash_default_tshow_flash_default;
    void   ifs_show_flash_detailed_tshow_flash_detailed;
    void           ifs_show_flash_errshow_flash_err;
    void     ifs_show_flash_filesys_tshow_flash_filesys;
    void     ifs_show_flash_summary_tshow_flash_summary;
} ifs_show_flash_vector;
```

These hooks should be set with the `ifs_fsid_set_show_flash_vector()` routine.

A flash file system may supply a data structure containing any or all of these hooks. IFS will supply a "show <fsname>" command for each flash file system contained in a router. If the file system has supplied the above vector, and non-null entry in the vector will cause the appropriate command option to appear in the show flash command for that file system. This allows IFS to restrict the available options to those that are actually pertinent to the file system in question. A C7000, for example, will present different options for the "show flash:" and "show slot0:" commands because these two file systems use different flash drivers.

9.6 References

ISO/IEEE 9945-1 1996 (ANSI/IEEE Std 1003.1, 1996 Edition) POSIX Part 1: System Application Program Interface (API) [C Language], ISBN 1-55937-573-6.

CHAPTER 10

Socket Interface

This Cisco IOS socket interface implements a subset of the standard UNIX socket functions. The Cisco IOS functions work over TCP and UDP, and perform identically or almost identically to their counterparts in the standard UNIX socket library. The function names are the same as those in the standard UNIX socket library, except that the string `socket_` has been added to the beginning of each name.

Table 10-1 lists the socket function calls supported by the Cisco IOS socket implementation, along with their standard UNIX equivalent.

Table 10-1 Cisco IOS Socket Functions and Macros

Function Name	Standard UNIX Function Name
socket_accept()	accept()
socket_bind()	bind()
socket_close()	close()
socket_connect()	connect()
socket_gethostbyname()	gethostbyname()
socket_get_localname()	getsockname()
socket_get_option()	getsockopt()
socket_get_peername()	getpeername()
socket_inet_addr()	inet_addr()
socket_inet_network()	inet_network()
socket_inet_ntoa()	inet_ntoa()
socket_listen()	listen()
socket_open()	socket() and open()
socket_recv()	recv()
socket_recvfrom()	recvfrom()
socket_select()	select()
socket_send()	send()
socket_sendto()	sendto()
socket_set_option()	setsockopt()
socket_share_fds()	None
socket_shutdown()	shutdown()
socket_strerror()	strerror()
socket_watch_other_events()	None

Function Name	Standard UNIX Function Name
Macro Name	
FD_CLR	
FD_ISSET	
FD_SET	
FD_SETSIZE	
FD_ZERO	

Note The following functions are yet to be documented in the *Cisco IOS API Reference*. In the meantime, please refer to the standard UNIX socket library for information about them: `socket_gethostbyname(), socket_inet_addr(), socket_inet_network(), socket_inet_ntoa(), socket_recvfrom()`, and `socket_sendto()`.

For information about the standard socket interface, refer to the following publications:

- Richard Stevens, *UNIX Network Programming*
- Douglas Comer *Internetworking with TCP/I* , Volumes I and III
- Richard Stevens and Gary Wright, *TCP/IP Illustrated*, Volumes 1 and 2

CHAPTER 11

ANSI C Library

Starting with Release 11.2, the Cisco IOS software is formalizing its use of ANSI C library functions. The “ANSI C Library” chapter in the *Cisco IOS API Reference* describes these library functions.

PART 3

Kernel Support Services

Subsystems

This chapter describes the programming interface and developer hooks for defining subsystems.

12.1 Overview: Subsystem

Subsystems provide independent entry points into the system code. They can be independent of the linker, or they can be freestanding code or part of code that always links and runs together. Subsystems allow images to be compiled that have the minimum of link requirements. Each subsystem itself is a discrete code module that supports various functions of an embedded system.

12.1.1 Subsystem Classes

Subsystems are organized into classes. Classes provide a sorting order, which is primarily used when initializing the system software. This is the only difference between classes. In all other respects, they provide the same functionality.

Currently, the following subsystem classes exist. These are listed in the order in which they are started when the system code is initialized.

- Registry
- Kernel
- Driver
- Protocol
- Library
- Management

Most subsystems currently are in the driver and protocol classes. In the future, some kernel subsystems (such as console drivers) and library subsystems (such as media encapsulation and de-encapsulation) will be created.

12.1.2 How to Choose a Subsystem Class

The class of subsystem you choose for your code depends primarily on when it should be initialized when the platform starts up. For example, picking a kernel or driver subsystem class reflects the fact that the subsystem provides elements that are intrinsically part of the running system. In general, the more abstract a function the subsystem provides to the system (and the further up in the protocol stack it resides), the later it should be initialized.

12.2 Subsystem Properties

Subsystem properties specify initialization dependencies. There are currently two types of properties:

- Sequencing—Defines the sequence in which subsystems must be initialized
- Requirements—Defines the other subsystems required by this subsystem

The subsystems listed in the sequencing and requirements properties do not need to be similar to each other nor do they need to be a subset of one another. In fact, they can be completely different if the subsystem does not care about the initialization order of the subsystems it requires.

12.2.1 Subsystem Property Definitions

Subsystem properties are defined in a header file. A subsystem property consist of a property identifier followed by one or more subsystem names. The property identifiers are `seq:` for sequencing properties and `req:` for requirements properties. The names are separated by white space or commas. White space in and around names is ignored when parsing individual items.

The subsystem definition can contain one or two property lists. If it has two lists, the two can be different (that is, one sequencing and one requirements list) or they can be two for the same property. For example, you can specify two sequencing property lists. If this case, the two lists are concatenated to form a single property list. The order in which you specify property lists does not matter.

If a subsystem does not need to use a `seq:` or `req:` property string, specify `NULL` for these properties. Do not specify `seq:` or `req:` by themselves with no subsystem name. This causes the code to do extra work to find out that there is nothing to process.

12.2.1.1 Subsystem Property Definitions: Examples

The following example defines a sequencing property for the subsystem stating that if the `sub1`, `sub2`, or `sub3` subsystems are present, they must start before this subsystem:

```
"seq: sub1, sub2, sub3"
```

The following example defines a requirements property for the subsystem that states that `sub1` must be present in order for this subsystem to operate:

```
"req: sub1"
```

12.2.2 Sequencing Property

The sequencing property defines the sequence in which subsystems must be initialized. You use this property to list the subsystems that must be initialized before the current subsystem can be initialized. All the subsystems in this list do not have to be present, but if they are present, they must start first.

For example, the `ip` protocol subsystem must be present in order for the `ipserver` protocol subsystem to function.

Sequencing properties need mention only subsystems that are in the same class. This is because the subsystem class structure itself dictates the larger granularity of subsystem initialization order when the system starts up. The order of subsystem class startup is as follows:

1 `SUBSYS_CLASS_REGISTRY`

2 SUBSYS_CLASS_KERNEL

3 SUBSYS_CLASS_DRIVER

4 SUBSYS_CLASS_PROTOCOL

5 SUBSYS_CLASS_LIBRARY

6 SUBSYS_CLASS_MANAGEMENT

This means, for example, that creating a SUBSYS_CLASS_MANAGEMENT subsystem that has a seq: property that references a subsystem of a class of SUBSYS_CLASS_PROTOCOL makes no sense. Although referencing another subsystem class effectively does no harm, it takes CPU time to process this dependency and is redundant.

The subsystem code displays a message if it finds cross-class sequence dependencies when the subsystems are being initialized. You should remove these extraneous dependencies from the code.

Do not overuse the sequencing property when creating subsystems. A large sequence list is a very strong indicator of a broken initialization structure. Referencing many subsystems in a seq: property forces the sequence list to be changed whenever a new subsystem is added to the system, which might disrupt the initialization order. Extensive referencing between subsystems does not scale and should be avoided.

12.2.3 Requirements Property

The req: property defines the subsystems that must be present in order for this subsystem to operate. All the subsystems listed must be present. The subsystems listed in the requirements property can be in any subsystem class.

12.2.4 Error Messages

When starting the system, the subsystem code checks the header structures that it finds to make sure that they are valid before initialization. If a subsystem specifies another subsystem in a req: that cannot be found, the following message is logged. The subsystem code will ignore the subsystem with the incomplete requirement dependency and will remain unstarted.

```
SUBSYS-2-NOTFOUND: Subsystem (xxx) needs subsystem (yyy) to start
```

The following messages can be produced by the code if a subsystem header is broken. A subsystem header contains information about the revision of system software it was compiled with and the format of the subsystem header it uses. This information is checked, and bounds on the subsystem class are also checked. Any deviation or inconsistency will cause the subsystem code to log an error These messages are rare and require serious investigation into the cause.

```
SUBSYS-2-BADVERSION: Bad subsystem version number (4) - ignoring subsystem
SUBSYS-2-MISMATCH: Kernel and subsystem version differ (10.2) - ignoring subsystem
SUBSYS-2-BASCLASS: Bad subsystem class (10) - ignoring subsystem
```

12.3 Define a Subsystem

You define a subsystem in a subsystem header definition by calling the SUBSYS_HEADER macro. In this macro, you define the entry point to the subsystem and the subsystem class. You also define any other subsystems on which this system depends.

```
SU BSYS_HEADER(char*name, ul o nmajor_version, u l o minor_version, u l o edit_version,
               vo id*init, ul o nclass, c har*property1, c har*property2);
```

The subsystem header definition must appear in the group of source files that represent the subsystem module. The header definition must be compiled into the data segment because that is the block of memory scanned for the subsystem information.

Every subsystem has an `init` routine or entry point. Subsystem entry points take a parameter. The entry point has the following form:

```
xxx_init (subsystype *subsys)
```

The pointer to the subsystem being initialized is passed through. This can allow the subsystem initialization routine to make the decisions about subsystem startup rather than using the default rules, which are governed by the `req:` property

12.3.1 Examples: Define a Subsystem

The following example defines a driver subsystem named `snark`, which is initialized by calling the entry point `snark_subsys_init()`. This is an example of a subsystem that can be initialized in a random order without needing any other subsystems to be present.

```
# d   efine SNARK_MAJVERSION1
# d   efine SNARK_MINVERSION0
# de  fine SNARK_EDITVERSION1

SUBSYS_HEADER (snark, SNARK_MAJVERSION, SNARK_MINVERSION, SNARK_EDITVERSION,
               snark_subsys_init, SUBSYS_CLASS_DRIVER, NULL, NULL);
```

The following example defines a driver subsystem called `snark`, which is initialized by calling the entry point `snark_subsys_init()`. The `seq:` portion of the macro indicates that when the code is being initialized, the `snark` subsystem must be initialized after the `boojum` and `wibble` subsystems. The `req:` portion of the macro indicates that the `snark` subsystem requires the `boojum` subsystem to be present for `snark` to work properly.

```
# d   efine SNARK_MAJVERSION1
# d   efine SNARK_MINVERSION0
# de  fine SNARK_EDITVERSION1

SUBSYS_HEADER (snark, SNARK_MAJVERSION, SNARK_MINVERSION, SNARK_EDITVERSION,
               snark_subsys_init, SUBSYS_CLASS_DRIVER, “seq: boojum, wibble”,
               “req: boojum”);
```

12.4 Fill In the Subsystem Structure

`SUBSYS_HEADER` is a macro definition that fills in the portions of the subsystem structure that are visible to the developer. The following is the subsystem structure:

```
struct subsystype_ {
    ulong magic1;
    ulong magic2;
    ulong header_version;
    ulong kernel_majversion;
    ulong kernel_minversion;
    char *namestring;
    ulong subsys_majversion;
    ulong subsys_minversion;
    ulong subsys_editversion;
    subsys_init_type *init_address;
    ulong class;
    ulong ID;
    char *properties[SUBSYS_PROPERTIES_MAX];
};
```

The variables `magic1` and `magic2` comprise a 64-bit magic number that is used to find the subsystem headers in the data segment.

The `kernel_majversion` and `kernel_minversion` variables define the kernel version levels. The system sets these levels when it compiles the module. When the code starts running, the system checks the version levels again to ensure that the subsystem is the correct version to run with the kernel.

The `ID` variable is a unique numeric value that is assigned to each subsystem when it is discovered by the kernel.

12.5 Tips for Creating a Subsystem

This section discusses the following programming tips for creating and working with subsystems:

- Create a New Subsystem
- Rework System Processes
- Reexamine Header File Dependencies
- Use New IDB Subblocks to Store Private Variables

12.5.1 Create a New Subsystem

To determine whether to create a new subsystem, follow these steps:

Step Identify a logical unit of functionality. A logical unit is one that is strongly cohesive, that is, the operations within the unit are closely related.

Step Determine whether each unit should be considered part of the core system or made into a separate and dependent subsystem. Generally, a unit should be made into a subsystem if its functionality is not required for basic operation and the unit is loosely coupled with others. For example, starting with Cisco IOS Release 11.1, AppleTalk Enhanced IGRP has been divided out into a separate subsystem called ATEIGRP, and MacIP, AURP, and IPTalk have been placed into another subsystem called ATIP.

For each unit that you decide to make into a subsystem, consider the following:

- Decide which functions and data variables belong in the new subsystem. These items might need to be relocated into a common set of files.
- Include the parse chains for the feature in the subsystem. These include, but are not limited to, global, interface, **debug**, and **show** commands. The commands are linked into the parser chain through the `parser_extension_request` array, which is passed to the `parser_add_command_list()` function. To minimize the proliferation of files, combine all the parse chains into one *xxx*`_chain.c` file. For information about dynamically adding commands to existing parse chains, see the "Command-Line Parser" chapter.
- Check for reasonable subsystem dependencies. Make sure you are aware of the other subsystems that are required when you establish a dependency to another subsystem. For example, if your subsystem requires the `IPSERVICES` subsystem, you should be aware that `IPSERVICES` requires `IPHOS` . Gwynne Franzino has developed a tool that generates a dependencies report to assist you. For more information, see `http://wwwin-swtools/~gwynne`.
- Define the new subsystem by declaring the subsystem header, `SUBSYS_HEADER`. You need to specify subsystem dependencies in this header. You can optionally define a subsystem initialization routine that is called once at system startup. This routine typically allocates required memory, adds the subsystem's service routines into various registries, and calls any subsubsystem initialization routines, such as debug and parser support. For more information, see the "Fill In the Subsystem Structure" section in this chapter.

  The subsystem header consumes 14 longwords.
- Rework the `makefile`. You need to define the new subsystem in the relevant `makefile` and `makesubsys` files. If you are defining a new subset image, you need to modify `makeimages`. See these files for examples of how this is done. Follow by example.
- Use registries judiciously. Calls from fully dependent subsystems into the core system generally do not need to use registries, but you get bonus points if you do. Calls from the core out to the subsystem do need to use a registry. When establishing hooks into a subsystem strive for a minimal but complete interface. Avoid stubs at all costs. For more information about registries, see the "Registries and Services" chapter.

  Each service created in a registry consume the following amount of run-time memory:

  — Case service with a case table: (2 * case table size) + 13 longwords

  — All other services: 13 longwords + 2 longwords per added service

  For example, the AppleTalk registry incurs the following overhead:

Service	Number of Longwords
List with 4 service routines	21
List with 1 service routine	15
List with 1 service routine	15
List with 1 service routine	15
List with 2 service routines	17
List with 2 service routines	17
List with 1 service routine	15
List with 2 service routines	17
List with 4 service routines	21
List with 1 service routine	15
List with 2 service routines	17

Service	Number of Longwords
List with 2 service routines	17
List with 2 service routines	17
Retval with 1 value	15
Stub	15
Stub	15
Loop with 1 service routine	15
Loop with 1 service routine	15
TOTAL	**294**

- Decide whether operations within the subsystem should be managed by a separate system process.
- The following is a suggested organization for files, where *xxx* is the name of the subsystem:
  - *xxx*`_chain.c`: Parser chain support
  - *xxx*`_init.c`: Subsystem header and initialization
  - *xxx*`_debug.c`, *xxx*`_debug.h`: Debug support

- — *xxx*_parse.c: Parser actions support
- — *xxx*_*: Actual subsystem files

12.5.2 Rework System Processes

You must rewrite all older systems processes to use the new Cisco IOS event-driven scheduler primitives.You can now set up processes to be driven by a large class of events, including managed timers, semaphores, signals, and messages. Rewriting these processes increases overall system performance. For an example, see the Banyan VINES code.

For information about processes and the scheduler, see the "Scheduler" chapter.

12.5.3 Reexamine Header File Dependencies

Including many header files in files is often redundant.To eliminate unnecessary interdependencies between files, remove header files when possible. Be sure, however, to build all subset images to determine you have not removed required header files.

12.5.4 Use New IDB Subblocks to Store Private Variables

It is no longer necessary or desirable to store private variables in the main IDB structure. Use subblocks instead. For more information, see the "Interfaces and Drivers" chapter. For an example, see the Banyan VINES code.

CHAPTER 13

Registries and Services

This chapter describes the programming interface and operation of Cisco IOS registries and services.

13.1 Overview: Registries and Services

Registries and services form a generic, linker-independent mechanism that permits subsystems to install or register callback functions, discrete values, or process IDs for a service provided by the Cisco IOS kernel or other modules.

A *registry* is a collection of *services* and is used as a container to hold services for a similar functional area (such as IP, AppleTalk, or X.25). These services provide an interface into a subsystem that is independent of linker relationships. This design allows subsystems to be compiled independently into an image but still be able to access services in another subsystem when both are present.

Services can be one of eight different types. In its simplest form, a service can be thought of as a managed function vector. However, the real power available through services comes from the ability to declare the function call semantics of a particular service invocation. These semantics are unique to each service type and allow common C constructs to be emulated in a generic and extensible way. For example, a case service point allows a `switch()` statement to be built dynamically, and a loop service allows a `while()` loop to be likewise emulated.

By allowing these service points to be defined and grown dynamically at run time, registry and service clients can build extensible code hooks that allow new protocols and features to be integrated into the existing code base with the minimal amount of disruption.

13.2 Registry Compiler: Description

The engine of the registry code is simple and entirely generic. In order to define registries and services, a *registry compiler* is used to compile a registry definition into several files that is used during the build process to provide prototypes and definitions. The chief job of the registry compiler is to provide wrapper functions for registry services that force full typechecking of registry call invocations. This typechecking is essential to prevent programming errors from producing subtle, elusive, and ultimately catastrophic bugs.

13.3 Registry Files

Table 13-1 describes the registry files associated with the Cisco IOS registries. Youcreate some of these files, and others are created by the registry compiler.

Table 13-1 Registry Files

File Suffix	Source	Contents
`.reg`	Created by programmer	Actual definitions for a registry and the services provided by it. This file is compiled by the registry compiler to create the `.regc` and `.regh` files. The `.reg` file is under source control.
`.regh`	Autogenerated by registry compiler	All the wrapper functions for registry services. This file is autogenerated by the registry compiler. All clients of the registry use these wrappers to add, delete, and change functions, values and PIDs for services. This file also provides the wrappers for service invocation. Users of the registry module must not `#include` this file directly. The `.regh` file is not under source control.
`.regc`	Autogenerated by registry compiler	Initialization code for the registry and its services. The owner of the actual directory `#includes` and executes the `.regc` file. No user modules except the `xxx_registry.c` may `#include` the `.regc` file. Ignoring this restriction can cause bizarre image problems. This file is not under source control.
`.h`	Created by programmer	User interface to clients of a given registry. This file `#includes` the generated `.regh` and all prerequisite `.h` files, declares the `reg_invoke`, `reg_add`, and other registry functions for each service point in the registry. The clients of the registry must `#include` this `.h` file, not the `.regh` file. Clients include the `.c` files that contain the registry initialization code and any `.c` files that need to acceess the registry service points in registry module.
`.c`	Created by programmer	Registry initialization code, of a fixed pattern. Each registry module is a separate subsystem of the `SUBSYS_CLASS_REGISTRY` class, thus ensuring that the module is initialized prior to its use. The `.c` file `#includes` the registry `.h` and `.regc` files.

13.4 Registry Compilation Process

The generic registry and service handling code does not perform strong typecasting on the parameters passed through it, because it has no knowledge of the actual service itself. Therefore, some form of protection is required to prevent errant code from passing incorrect parameters. This protection is achieved by automatically building wrappers around the registry addition, deletion, and invocation functions that are used to manipulate the services. In order to build these wrappers, which take the form of inline functions, the services that make up a registry are described in an intermediate metalanguage in the `.reg` file.

13.4.1 11.3 Changes

In 11.3, the back end of the registry compiler was completely rewritten, and generates entirely different code from the previous version. The fundamental difference is that each registry module now provides an independent global structure instance with a single extern symbol for the linker to resolve.

13.4.1.1 registry.c

The concept of generating inline code to access generic services in `registry.c` is retained, but the generated code is considerably smaller, and provides many of the services without calling on helper routines in `registry.c`. Where helper routines are called, they are called with fewer parameters than previously, and the helper routines themselves are smaller. For instance, of the `reg_invoke` inlines, neither `STUB`, `LIST` nor `FASTCASE` service points use helper routines, while `CASE`, `RETVAL` and `PIDLIST` do.

13.4.1.2 registry.h

The old support had an `enum` in `h/registry.h`. Each registry module had an entry in this enumeration. This `enum` no longer exists, and is no longer necessary. This means that `registry.h` no longer contains a knowledge of all the registry modules in the system, and so has now become a stable interface. It reflects only the design of the registry infrastructure, and not the users of it.

In fact, none of the content of `registry.h` is to be referenced by user code. The only references to it should be from registry-compiler-generated code. It now defines only structures and inlines internal to the registry infrastructure.

13.4.1.3 Static and dynamic registries:

There is no longer any concept of static and dynamic registries. Each registry module is either referenced in an image or not, and is now of a minimal size so that none of it can be regarded as wasted space. If it is referenced then it will be included in the image. Along with this is the dropping of the requirement to call `create_registry(REG_MODULE_ENUM)`, and the removal of the need to remember a dynamic regcode value. The registry modules are now identified solely by a single name through the linker. This is a name like `_registry_xxx` and is a global instance of a struct containing everything necessary to support the service points in that registry module. The leading '_' is an indicator that this name is for internal use by the registry infrastructure, and is not to be referenced by user code.

13.4.1.4 Generated Code

The generated code now provides for stronger type checking by the compiler of the provided parameters in `reg_` calls, especially for the `FAST_CASE` registry type.

Most of the `FAST` service types are now identical with their non-`FAST` counterparts. `FASTCASE` is the one remaining different one. It remains faster than `CASE` by virtue of avoiding the range check.

For more details of the generated code, see any of the generated `.regh` files, eg `atm/atm_registry.regh`.

13.5 .reg File Metalanguage

The metalanguage used in the `.reg` file follows these formatting rules:

- Place each item on its own line.
- To continue a line, end it with a backslash (\).
- Begin comment lines with pound sign (#).
- Name the registry to be created in the `BEGIN REGISTRY` statement. The name of the registry must be in all uppercase letters. For example:

```
BEGIN REGISTRY REGISTRY_NAME
```

- Terminate the definition of a registry with an `END REGISTRY` statement.
- Define each service point, along with its attributes, between a `DEFINE` and an `END` statement. The name of the service point must be in all lowercase letters. For example:

```
DEFINE service_point_name
.
.
.
END
```

- Position items within the `DEFINE` statement as follows:
  - The first item is a required comment, specified in standard C format. For example:

    ```
    /* comment */
    ```

    The comment is reformatted to fit the output lines unless it is written in comment bar format. Comment bars are copied as is. For example:

    ```
    /*
     * comment
     */
    ```

  - The next item is an optional `DATA` block. All text between `DATA` and `END DATA` is copied and placed between the comment bar and the function declaration. This text is used to include additional types that are required by the function definition.
  - The next item is the type of service. It must be `CASE`, `LIS` , `LOO` , `PID_LIS` , `RETVAL`, `STUB`, or `VALUE`. For definitions of the service types, see the section "Types of Services" in this chapter.
  - Next is the type declaration of the value returned by the called function. It must be `void` for `LIS` , `PID_LIS` , and `CASE` services. It must be `boolean` for `LOOP` services. It must be `ulong` for `VALUE` services.

    For a `VALUE` service, no additional parameters can be specified. For all other services, you specify the prototype list for the called function. If a particular service requires no parameters, use a hyphe n(-).

    `CASE`, and `RETVAL` services require two additional parameters. The first is the number of cases for the case registry, and the second is the type for the variable specifying the case.
  - The last item, which is present for `PID_LIST` services only, is the message identifier to be sent to each process.

13.5.1 Example: .reg File Format

The following is an example of a `.reg` input file:

```
BEGIN REGISTRY SAMPLE
DEFINE sample_service1
/*
 *A list service
 */
  LIST
  void
  -
END

DEFINE sample_service2
/*
 * A loop service that requires a structure definition
 */
DATA
  typedef struct boojum_ {
    int a;
    int b;
  } boojum;
END DATA
LOOP
  boolean boojum *snark, int delta
END

DEFINE sample_service3
/*
 *A stub service
 */
STUB
  void
  int count, char *name
END

DEFINE sample_service4
/*
 * A case service
 */
CASE
  void
  boolean onoff, int no_packets
  MAX_CASES
  ushort protocol
END

DEFINE sample_service5
/*
 * A retval service
 */
RETVAL
  char * int errors, int drops, int collisions, int transmits
  MAX_CASES
  ulong media
END

DEFINE sample_service6
/*
 * A value service
 */
VALUE
  ulong
END
```

```
DEFINE sample_service7
/*
 * A pid_list service
 */
PID_LIST
  void
  idbtype swidb
  MSG_SERVICE7
END

END

END REGISTRY
```

13.6 .h File Contents

The following is an example of the contents of the `.h` file for a registry module. For the `.regh` file to compile, you must declare the parameter types and define the size of any case registries.

```
#ifndef __SAMPLE_REGISTRY_H__
#define __SAMPLE_REGISTRY_H__

#include “registry.h”

#include “sample_registry_prereqs.h”

#include “sample_registry.regh”

#endif
```

13.7 .c File Content

The following is an example of the contents of the `.c` file for a registry module. In this example, the registry subsystem initialization calls `create_registry_sample()`, which initializes the structures generated by the registry compiler to support the defined registry services.

```
#include “master.h”
#include “subsys.h”
#include “sample_registry.h”
#include “sample_registry.regc”

/*
 * sample_registry_init
 *
 * Initialize sample registry.
 */

static void sample_registry_init (subsystype *subsys)
{
create_registry_sample();
}

/*
 * Sample Registry subsystem header
 */
#define SAMPLE_REGISTRY_MAJVERSION 1
#define SAMPLE_REGISTRY_MINVERSION 0
#define SAMPLE_REGISTRY_EDITVERSION 1

SUBSYS_HEADER(sample_registry,
              SAMPLE_REGISTRY_MAJVERSION, SAMPLE_REGISTRY_MINVERSION,
              SAMPLE_REGISTRY_EDITVERSION,
              sample_registry_init, SUBSYS_CLASS_REGISTRY,
              NULL, NULL);
```

13.8 Placement of xxx_registry.o in makefiles

Former usage often packaged the `xxx_registry.o` modules with subsystems which were `reg_add` clients of the xxx registry.

It is now required to package them where they will be included in images with *all* clients - whether `reg_add`, `reg_invoke` or other. With well-designed registry usage, this typically means that they would be packaged with the subsystem which contains `reg_invoke` calls for the registry functions. With not-so-well designed registry modules, this can be very difficult to figure out. To solve this problem, a registry library has been set up. Placing a registry module in the registry library ensures that it will be included in all images where it is needed.

Because of the irregular content of existing registry modules, and the difficulty of locating a single generic subsystem which would be a suitable home for each existing registry module, the conversion effort placed all registry modules in the registry library.

If registry owners find it possible to place some of them more explicity in suitable subsystems, they can be removed from the registry library. Some effort of this nature is already occurring in the 11.3 and later codebases.

13.9 Services: Overview

A *service* is a data structure that describes how a collection of one or more C functions, discrete values, or process IDs should be handled when the service is invoked by a service client. All members of a specific service have the same properties, such as calling and return conventions. The actual instance of a service is referred to as a *service point.*

For example, the `REG_SYS` registry supports the `SERVICE_RAW_ENQUEUE` service, which allows a driver module to enqueue a datagram destined for the router onto a particular protocol input queue. When a protocol subsystem initializes itself, one of the functions it registers is the protocol-specific enqueuing function. If that protocol is not present in the system, nothing is registered, and a default action occurs when the service is invoked with a datagram belonging to that protocol. In this case, the datagram is quietly discarded.

13.10 Types of Services

The registry support provides the following types of services. The names refer to the way in which the registered functions are invoked.

- List. A list service is the run-time replacement for a list of C functions that are executed sequentially. It reads through a list of functions, calling one function at a time.
- Pid_list. A pid_list service is similar to the list service except that it reads through a list of process identifiers, sending the same message to each process.
- Case. A case service is a run-time replacement for a C `switch` statement. It reads through a list of functions until it finds the matching service.
- Retval. A retval service is identical to the case service except that it returns a value instead of a `void`.
- Loop. A loop service is a run-time replacement for a C `while` loop. Each function registered for the loop service is called until one of the functions returns `TRUE`.
- Stub. A stub service takes zero or one functions (like a list service) and can return a value (like a retval service).
- Value. A value service is a lookup table of 32-bit values.

13.11 'show registry' Support

The format of the **show registry** command output changed in 11.3 to reflect the new internal structure. This is not fully described here, but is largely self-explanatory when the functionality of the different registry types is understood.

The content of the data structures for each registry service point is printed. In the printout, hex addresses are always the addresses of routines which have been installed in the corresponding service point. These addresses are most usefully cut and pasted into an `rsym` input window, to show the real function names.

The service point numbers are listed in the **show registry** output, and these can be correlated manually with the service point names through the table generated at the top of the relevant `xxx_registry.regh` file. Service points appear in this list sorted first by type of service point, and then by order of appearance in the definition `xxx_registry.reg` file.

13.12 Manipulate List Services

A list service is the run-time replacement for a list of C functions that are executed sequentially. When a list service is invoked, it reads through a list of functions, calling one function at a time.

13.12.1 Define a List Service

To define a list service for a registry, use the following syntax:

```
DEFINE name
LIST
void
arguments
END
```

The return value from a list service is always `void`.

On the `arguments` line, you specify the arguments for the service, and, as a consequence, the prototype of the functions called.

From the list service definition, the registry compiler generates all the wrappers needed to manipulate the list service. The following wrapper functions are generated for a list service:

```
void reg_invoke_name(arguments);
void reg_add_name(service_name_type callback, char *textual_name);
void reg_delete_name(service_name_type callback);
```

The registry compiler substitutes `name` in the above functions and in prototype names for the service name declared on the `DEFINE` line of the service definition. The registry compiler also uses the `arguments` declared in the service definition for the prototype of the invocation wrapper and callback.

13.12.1.1 Example: Define a List Service

The following is an example of a list service definition. This service is called when the fast-switching state is initialized for an interface. It takes only one argument, which is a pointer to a hardware IDB.

```
DEFINE fast_setup
LIST
void
hwidbtype *hwidb
END
```

This list service definition generates the following wrappers:

```
void reg_invoke_fast_setup(hwidbtype *hwidb);
void reg_add_fast_setup(service_fast_setup_type callback, char *name);
void reg_delete_fast_setup(service_fast_setup_type callback);
```

13.12.1.2 Example:Add To a List Service

When adding a list service for the `fast_setup` service, the registry compiler takes the parameters from the `reg_add_fast_setup()` function and uses them to generate a strongly typecast wrapper:

```
typedef void (*service_fast_setup_type) (hwidbtype *hwidb);

#define reg_add_fast_setup(a,b) _reg_add_fast_setup(a)
static inline void _reg_add_fast_setup (service_fast_setup_type callback)
{
    registry_add_list(callback, &_registry_sys.fast_setup);
}
```

The following code uses the list service addition wrapper to add the `atalk_fast_setup()` function to the `fast_setup` service:

```
reg_add_fast_setup(atalk_fast_setup, “atalk_fast_setup”) ;
```

Then, whenever the `fast_setup` service is invoked, the `atalk_fast_setup()` function is called.

13.12.1.3 Example: Invoke a List Service

When invoking a list service for the `fast_setup` service, the registry compiler takes the parameters from the `reg_invoke_fast_setup()` function and uses them to generate a strongly typecast wrapper:

```
static inline void reg_invoke_fast_setup (hwidbtype *hwidb)
{
    reg_list_struct *list = _registry_sys.fast_setup;
    while (list) {
    (*(service_fast_setup_type)list->function) (hwidb);
    list = list->next;
    }
}
```

The following code illustrates how to invoke all the functions registered for the `fast_setup` service:

```
reg_invoke_fast_setup(hwidb);
```

13.13 Manipulate Pid_list Services

A pid_list service is similar to a list service in that it is used as the run-time replacement for a list of C functions that are executed sequentially. This service reads through a list of process identifiers, sending the same message to each process. The advantage of using a pid_list service over a list service is that the messages are received and all processing is performed on the recipient’s stack and in the recipient’s execution thread. This eliminates problems caused by multiple execution threads accessing the same data structures.

13.13.1 Define a Pid_list Service

To define a pid_list service for a registry, use the following syntax:

```
DEFINE name
PID_LIST
void
arguments
msgtype
END
```

The return value from a pid_list service is always `void`.

On the *`arguments`* line, you specify the arguments for the service, and, as a consequence, the prototype of the functions called.

On the *`msgtype`* line, you specify the message minor event to pass through with the process handler.

From the pid_list service definition, the registry compiler generates all the wrappers needed to manipulate the list service. The following wrappers are generated for a pid_list service:

```
void reg_invoke_name(arguments);
void reg_add_name(pid_t pid, char *textual_name);
void reg_delete_name(pid_t pid);
```

The registry compiler substitutes *`name`* in the above functions and in prototype names for the service name declared on the `DEFINE` line of the service definition. The registry compiler also uses the *`arguments`* declared in the service definition for the prototype of the invocation wrapper and callback.

13.13.1.1 Example: Define a Pid_list Service

The following is an example of a pid_list service definition. This service is called when an interface state changes to allow routing protocols to adjust their internal routes. It takes only one argument, which is a pointer to a software IDB.

```
DEFINE route_adjust_msg
PID_LIST
void
idbtype *swidb
MSG_ROUTE_ADJUST
END
```

This pid_list service definition generates the following wrappers:

```
void reg_invoke_route_adjust_msg(idbtype *swidb);
void reg_add_route_adjust_msg(pid_t pid, char *name);
void reg_delete_route_adjust_msg(pid_t pid);
```

13.13.1.2 Example:Add To a Pid_list Service

When adding a pid_list service for the `route_adjust_msg` service, the registry compiler takes the parameters from the `reg_add_route_adjust_msg()` function and uses them to generate a strongly typecast wrapper:

```
#define reg_add_route_adjust_msg(a,b) _reg_add_route_adjust_msg(a)
static inline void _reg_add_route_adjust_msg (pid_t pid)
{
    registry_add_pid_list(&_registry_sys.route_adjust_msg, pid);
}
```

The following code uses the pid_list addition wrapper to add the `vines_rtr_pid` process to the `route_adjust_msg` service:

```
reg_add_route_adjust_msg(vines_rtr_pid, “vines_router”);
```

Then, whenever the `route_adjust_msg` service is invoked, the process with the PID given by `vines_rtr_pid` is sent a message of minor type `MSG_ROUTE_ADJUS` .

13.13.1.3 Example: Invoke a Pid_list Service

When invoking a pid_list service for the `vines_rtr_pid` process, the registry compiler takes the parameters from the `reg_invoke_route_adjust_msg()` function and uses them to generate a strongly typecast wrapper:

```
static inline void reg_invoke_route_adjust_msg (idbtype *swidb)
{
    registry_pid_list(&_registry_sys.route_adjust_msg, swidb);
}
```

The following codes illustrates how to send `MSG_ROUTE_ADJUST` messages to all the processes registered for the `route_adjust_msg` service:

```
reg_invoke_route_adjust_msg(swidb);
```

13.14 Manipulate Case Services

A case service is a run-time replacement for a C `switch` statement. This service reads through a list of functions until it finds the matching service.

13.14.1 Define a Case Service

To define a case service for a registry, use the following syntax:

```
DEFINE name
CASE
void
arguments
maximum
index
END
```

The return value from a case service is always `void`.

The prototype of the variable used to index the case service is defined on the *`index`* line. If *`maximum`* is nonzero, a lookup table is generated to allow faster indexing to function lookups. Faster indexing is performed at the expense of memory.

On the *`arguments`* line, you specify the arguments for the service, and, as a consequence, the prototype of the functions called.

From the case service definition, the registry compiler generates all the wrappers needed to manipulate the case service. The following wrapper functions are generated for a case service:

```
void reg_invoke_name(index, arguments);
void reg_add_name(index, service_name_type *callback, char *textual_name);
void reg_add_default_name(service_name_type *callback, char *textual_name);
void reg_delete_name(index, service_name_type *callback);
static inline void reg_delete_default_name(void);
boolean reg_used_name(index);
```

The registry compiler substitutes `name` in the above functions and in prototype names for the service name declared on the `DEFINE` line of the service definition. The registry compiler also uses the `arguments` declared in the service definition for the prototype of the invocation wrapper and callback.

13.14.1.1 Example: Define a Case Service

The following is an example of a case service definition for the `raw_enqueue` service. This service is called by the incoming encapsulation demultiplexer to allow packets to be routed to the correct protocol. It takes only one argument, which is a pointer to the packet to be handled.

```
DEFINE raw_enqueue
CASE
void
paktype *pak
LINK_MAXLINKTYPE
int linktype
END
```

This case service definition generates the following wrappers:

```
#define reg_add_raw_enqueue(a,b,c) _reg_add_raw_enqueue(a,b)
void _reg_add_raw_enqueue(int linktype, service_raw_enqueue_type callback);
#define reg_add_default_raw_enqueue(a,b) _reg_add_default_raw_enqueue(a)
void _reg_add_default_raw_enqueue(service_raw_enqueue_type callback);
void reg_delete_raw_enqueue(int linktype);
void reg_delete_default_raw_enqueue (void);
boolean reg_used_raw_enqueue(int linktype);
void reg_invoke_raw_enqueue(int linktype, paktype *pak);
```

13.14.1.2 Example: Add a Case Service

When adding a case service for the `raw_enqueue` service, the registry compiler takes the parameters from the `reg_add_raw_enqueue()` function and uses them to generate a strongly typecast wrapper:

```
typedef void (*service_raw_enqueue_type) (paktype *pak);

#define reg_add_raw_enqueue(a,b,c) _reg_add_raw_enqueue(a,b)
static inline void _reg_add_raw_enqueue (int linktype,
                                         service_raw_enqueue_type callback)
{
    registry_add_case(linktype, callback, &_registry_sys.raw_enqueue);
}
```

The following codes uses the case service addition wrapper to add the `etalk_enqueue()` function to the `raw_enqueue` case service:

```
reg_add_raw_enqueue(LINK_APPLETALK, etalk_enqueue, “etalk_enqueue”);
```

Then, whenever the `raw_enqueue` service is invoked with an index of `LINK_APPLETALK`, the `etalk_enqueue()` function is called.

13.14.1.3 Example: Add a Default Case Function

When adding a default case service for the `raw_enqueue` service, the registry compiler takes the parameters from the `reg_add_raw_enqueue()` function and uses them to generate a strongly typecast wrapper:

```
#define reg_add_default_raw_enqueue(a,b) _reg_add_default_raw_enqueue(a)
static inline void _reg_add_default_raw_enqueue (service_raw_enqueue_type callback)
{
    _registry_sys.raw_enqueue.default_function = callback;
}
```

The following code illustrates how to add the `netinput_enqueue()` default function to the `raw_enqueue` case service:

```
reg_add_default_raw_enqueue(netinput_enqueue, “netinput_enqueue”);
```

Whenever the `raw_enqueue` service is invoked with an index that has no function explicitly bound to it, `netinput_enqueue()` is called. This behavior mimics the default case in a C `switch` statement.

13.14.1.4 Example: Invoke a Case Service

When invoking a case service for the `raw_enqueue` service, the registry compiler takes the parameters from the `reg_invoke_raw_enqueue()` function and uses them to generate a strongly typecast wrapper:

```
static inline void reg_invoke_raw_enqueue (int linktype, paktype *pak)
{
    service_raw_enqueue_type function =
        registry_case(linktype, &_registry_sys.raw_enqueue);
    (*function) (pak);
}
```

The following code illustrates how to invoke a case service for a specified index. In this example, if a function is registered for the value of `pak->linktype`, the function is called when this statement is executed. If no function is called, the default function is executed. If no default is registered, the service returns without executing anything.

```
reg_invoke_raw_enqueue(pak->linktype, pak);
```

13.15 Manipulate Retval Services

A retval service is identical to a case service in its description, addition, default addition, and invocation. However, a retval service returns a value instead of `void`.

13.16 Manipulate Loop Services

A loop service is a run-time replacement for a C `while` loop. Each function registered for the loop service is called until one of the functions returns `TRUE`.

13.16.1 Define a Loop Service

To define a loop service for a registry, use the following syntax:

```
DEFINE name
LOOP
boolean
arguments
END
```

The return value from a loop service is always `boolean`.

On the *`arguments`* line, you specify the arguments for the service, and, as a consequence, the prototype of the functions called.

From the loop service definition, the registry compiler generates all the wrappers needed to manipulate the loop service. The following wrapper functions are generated for a loop service:

```
#define reg_add_name(a,b) _reg_add_name(a)
void _reg_add_name(service_name_type callback);
void reg_delete_name(service_name_type callback);
boolean reg_invoke_name(arguments);
```

The registry compiler substitutes `name` in the above functions and in prototype names for the service name declared on the `DEFINE` line of the service definition. The registry compiler also uses the `arguments` declared in the service definition for the prototype of the invocation wrapper and callback.

13.16.1.1 Example: Define a Loop Service

The following is an example of the loop service definition for the `ip_udp_input` service. This service is called to absorb a datagram. The first function to absorb the datagram returns `TRUE`, which prevents remaining functions from being called.

```
DEFINE ip_udp_input
LOOP
boolean
paktype *pak, udptype *udp
END
```

This loop service definition generates the following wrappers:

```
#define reg_add_ip_udp_input(a,b) _reg_add_ip_udp_input(a)
void _reg_add_ip_udp_input(service_ip_udp_input_type callback);
void reg_delete_ip_udp_input(service_ip_udp_input_type callback);
boolean reg_invoke_ip_udp_input(paktype *pak, udptype *udp);
```

13.16.1.2 Example:Add To a Loop Service

When adding a loop service for the `ip_udp_input` service, the registry compiler takes the parameters from the `reg_add_ip_udp_input()` function and uses them to generate a strongly typecast wrapper:

```
typedef boolean (*service_ip_udp_input_type) (paktype *pak, udptype *udp);

static inline void _reg_add_ip_udp_input (service_ip_udp_input_type callback)
{
    registry_add_list(callback, &_registry_ip.ip_udp_input);
}
```

The following code uses the loop service addition wrapper to add the `cayman_udp_decaps()` function to the `ip_udp_input` service:

```
reg_add_ip_udp_input(cayman_udp_decaps, “cayman_udp_decaps”);
```

Then, whenever the `ip_udp_input` service is invoked, the `cayman_udp_decaps` is one of a number of routines that has a chance to absorb the datagram. The first routine that does so returns `TRUE`, thus preventing the others being invoked.

13.16.1.3 Example: Invoke a Loop Service

When invoking the `ip_udp_input` service, the registry compiler takes the parameters from the `reg_add_parse_address()` function and uses them to generate a strongly typecast wrapper:

```
static inline boolean reg_invoke_ip_udp_input (paktype *pak, udptype *udp)
{
    reg_list_struct *list = _registry_ip.ip_udp_input;
    while (list) {
        if ((*(service_ip_udp_input_type)list->function) (pak, udp)) {
            return TRUE;
        }
        list = list->next;
    }
    return FALSE;
}
```

The following code illustrates how to invoke the `parse_address` loop service. If the datagram is absorbed by any of the registered functions, `TRUE` is returned, and the return statement is executed.

```
if reg_invoke_ip_udp_input(pak, udp)
    return;
```

13.17 Manipulate Stub Services

A stub service takes zero or one functions (like a list service) and can return a value (like a retval service).

Note In general, avoid using stub services. Instead, try to rework the software to use the more general case service instead.

13.17.1 Define a Stub Service

To define a stub service for a registry, use the following syntax:

```
DEFINE name
STUB
return
arguments
END
```

The *`return`* line specifies the return value from a stub service.

On the *`arguments`* line, you specify the arguments for the service, and, as a consequence, the prototype of the functions called.

From the stub service definition, the registry compiler generates all the wrappers needed to manipulate the stub service. The following wrapper functions are generated for a stub service:

```
void reg_invoke_name(arguments);
void reg_add_name(service_name_type *callback, char *textual_name);
void reg_delete_name;
```

The registry compiler substitutes *`name`* in the above functions and in prototype names for the service name declared on the `DEFINE` line of the service definition. The registry compiler also uses the *`arguments`* declared in the service definition for the prototype of the invocation wrapper and callback.

13.17.1.1 Example: Define a Stub Service

The following is an example of a stub service definition. This service is called to initialize the log facility in the router.

```
DEFINE log_config
STUB
void
parseinfo *csb
END
```

This stub service definition generates the following wrappers:

```
void reg_invoke_log_config(parseinfo *csb);
void reg_add_log_config(service_log_config_type *callback, char *name);
void reg_delete_log_config(void);
```

13.17.1.2 Example:Add To a Stub Service

When adding a stub service for the `log_config` service, the registry compiler takes the parameters from the `reg_add_log_config( )` function and uses them to generate a strongly typecast wrapper:

```
#define reg_add_log_config(a,b) _reg_add_log_config(a)

static inline void _reg_add_log_config (service_log_config_type callback)
{
    _registry_sys.log_config = callback;
}
```

The following code uses the stub service addition wrapper to add the `syslog_config()` function to the `log_config` service:

```
reg_add_log_config(syslog_config, “syslog_config”);
```

Then, whenever the `log_config` service is invoked, the `syslog_config()` function is called.

13.17.1.3 Example: Invoke a Stub Service

When invoking a stub service for the `log_config` service, the registry compiler takes the parameters from the `reg_invoke_log_config( )` function and uses them to generate a strongly typecast wrapper:

```
static inline void reg_invoke_log_config (parseinfo *csb)
{
    (*_registry_sys.log_config) (csb);
}
```

The following code illustrates how to invoke the function registered for the `log_config` service:

```
reg_invoke_log_config(csb);
```

13.18 Manipulate Value Services

A value service is a lookup table of 32-bit values. You can use this service to build a variety of sparse, dynamic lookup tables that can be filled by multiple code sections.

13.18.1 Define a Value Service

To define a value service for a registry, use the following syntax:

```
DEFINE name
VALUE
return
index
maximum
type
END
```

The *return* line specifies the return value from a value service.

The prototype of the variable used to index the value service is defined on the *index* line. If *maximum* is nonzero, a lookup table is generated to allow faster indexing to function lookups. Faster indexing is performed at the expense of memory.

From the value service definition, the registry compiler generates all the wrappers needed to manipulate the value service. The following wrapper functions are generated for a value service:

```
return reg_invoke_name(index);
void reg_add_name(index, type, char *textual_name);
void reg_add_default_name(type, char *textual_name);
```

The registry compiler substitutes *name* in the above functions and in prototype names for the service name declared on the DEFINE line of the service definition. The registry compiler also uses the *return* line declared in the service definition for the type of variable being registered for a given *index*.

13.18.1.1 Example: Define a Value Service

The following is an example of a value service definition. This service can be used to map ARPA type codes to internal packet link types.

```
DEFINE media_type_to_link
VALUE
ulong
ulong value
0
long type
END
```

This value service definition generates the following wrappers:

```
ulong reg_invoke_media_type_to_link(long type);
void reg_add_media_type_to_link(long type, ulong value, char *name);
void reg_add_default_media_type_to_link(ulong value, char *name)
```

13.18.1.2 Example:Add To a Value Service

When adding a value service for the media_type_to_link service, the registry compiler takes the parameters from the reg_add_media_type_to_link( ) function and uses them to generate a strongly typecast wrapper:

```
#define reg_add_media_type_to_link(a,b,c) _reg_add_media_type_to_link(a,b)

static inline void _reg_add_media_type_to_link (long type, ulong value)
{
    registry_add_value(type, value, &_registry_media.media_type_to_link);
}
```

The following example uses the addition wrapper to add the ARPA typecode for IP to the `media_type_to_link` service:

```
reg_add_media_type_to_link(TYPE_IP10MB, LINK_IP, “LINK_IP”);
```

Then, whenever the `media_type_to_link` service is invoked with an index of `TYPE_IP10MB`, the `LINK_IP` value is returned.

13.18.1.3 Example: Add a Default Value

When adding a default value service for the `media_type_to_link` service, the registry compiler takes the parameters from the `reg_default_media_type_to_link( )` function and uses them to generate a strongly typecast wrapper:

```
#define reg_add_default_media_type_to_link(a,b) \
    _reg_add_default_media_type_to_link(a)

static inline void _reg_add_default_media_type_to_link (ulong value)
{
    _registry_media.media_type_to_link.default_value = value;
}
```

The following example illustrates how to add a default value to a value service:

```
reg_add_default_media_type_to_link(LINK_ILLEGAL, “LINK_ILLEGAL”);
```

Whenever the `media_type_to_link` service is invoked with an index that has no value explicitly bound to it, `LINK_ILLEGAL` is returned.

13.18.1.4 Example: Invoke a Value Service

When invoking a value service for the `media_type_to_link` service, the registry compiler takes the parameters from the `reg_invoke_media_type_to_link( )` function and uses them to generate a strongly typecast wrapper:

```
static inline ulong reg_invoke_media_type_to_link (long type)
{
    return registry_value(type, &_registry_media.media_type_to_link);
}
```

The following example illustrates how to invoke a value service for a specified index. This function call attempts to find a valid link type for the ARPA type pointed to in the argument. If no value can be found for the given index, the default value is returned. If no default is declared, zero is returned.

```
pak->linktype = reg_invoke_media_type_to_link(ether->type_or_len);
```

Time-of-Day Services

14.1 Overview: Time-of-Day Services

The Cisco IOS software provides a rich set of time-of-day services. It contains a software time-of-day clock that can be interrogated and manipulated in various ways.

The time-of-day services are not intended to be used for simple periodic events, duration timing, and the like. For these functions, use the timer services described in the "Timer Services" chapter.

14.1.1 Epoch: Definition

An *epoch* is an instantaneous location in time, such as 3:15:35 p.m., Pacific Daylight Time, June 14, 1995. In the Cisco IOS software, the preferred form of an epoch is the `clock_epoch` structure, which is defined as follows:

```
typedef struct clock_epoch_ {
    ul            ong epoch_secs;/* Seconds */
    ul            ong epoch_frac;/* Fractional seconds */
} clock_epoch;
```

14.1.2 Time Formats

The Cisco IOS software uses three different time formats, which are referred to by the following names:

- `clock_epoch` structure
- UNIX format
- `timeval` structure

14.1.2.1 clock_epoch Structure

The time base for the `clock_epoch` structure is 0000 UTC, 1 January, 1900. (UTC is Coordinated Universal Time, which is also known as zulu time and was formerly known as Greenwich Mean Time [GMT]).This epoch does not include leap seconds, so the base actually shifts upon the addition and deletion of leap seconds.) The fractional part of the epoch is in units of 2^{-32} seconds, or approximately 0.2 nanoseconds, which is a very fine granularity.The integer seconds part of the timestamp will roll over sometime in the y ear2036.

14.1.2.2 UNIX Format

Some protocols used the so-called "UNIX format." It is stored as a 32-bit count of seconds since 0000 UTC,1 January, 1970. Such timestamps have poor granularity and are used in the Cisco IOS system code only minimally, where necessary.

14.1.2.3 timeval Structure

The `timeval` structure is a representation of an epoch broken up into hours, minutes, seconds, and so on. This structure also includes a time zone offset from UTC to support local time zones. It is defined as follows:

```
typedef struct timeval_ {
    ul             ong year;/* Year, AD (1993, not 93!) */
    ul            ong month;/* Month in year (Jan = 1) */
    ul              ong day;/* Day in month (1-31) */
    ul             ong hour;/* Hour in day (0-23) */
    ul           ong minute;/* Minute in hour (0-59) */
    ul           ong second;/* Second in minute (0-59) */
    ul      ong millisecond;/* Millisecond in second (0-999) */
    ul      ong day_of_week;/* Sunday = 0, Saturday = 6 */
    ul      ong day_of_year;/* Day in year (1-366) */
    lo         ng tz_offset;/* Time zone offset (seconds from UTC) */
} timeval;
```

14.1.3 System Clock: Description

The heart of the Cisco IOS time-of-day services is the *system clock*. It is a `clock_epoch` structure that is updated by hardware clock interrupts, advancing by an amount equal to the period of the hardware clock for each tick. This period is nominally 4 milliseconds, but it may vary slightly from platform to platform. Because the granularity of the system clock is so fine, the frequency of the clock can be varied with great precision by modifying the amount added for each tick. The frequency can be adjusted at a precision of roughly one part in 17 million.

Typically, the 4-millisecond granularity of the system clock is sufficiently accurate for most applications. However, if highly precise time is required, the Cisco IOS software can interpolate between hardware ticks by interrogating the hardware to find out how much time has elapsed since the last tick. This can improve the granularity of the time returned to a microsecond or better.

14.1.4 Time Zones

All epochs stored in `clock_epoch` structures are based on Coordinated Universal Time (UTC), which is also known as zulu time and was formerly known as Greenwich Mean Time (GMT). UTC is the time zone at zero degrees longitude.

Local time zones are often more convenient to work with, but they tend to be ambiguous.

Summer time, also known as daylight saving time, makes tracking local time even more challenging, because summer time rules vary from country to country, and even from state to state and county to county within some states in the United States. In fact, localities in the Southern Hemisphere that observe summer time do so at the opposite time of the year those north of the equator.

Within the Cisco IOS system code, time epochs are generally stored as UTC at all times except when times are displayed by the user interface.

14.1.5 Network Time Protocol

The Network Time Protocol (NTP) is closely tied to the maintenance of the Cisco IOS system clock. If NTP is enabled, it maintains the system clock to a very high degree of accuracy, adjusting the clock frequency to correct for the otherwise unavoidable drift caused by systematic errors in the clock hardware. NTP is by far the most preferable way of setting and maintaining the system clock, in addition to providing time service to other systems on the network.

14.1.6 Hardware Calendar

Some platforms support a clock/calendar in hardware. This is essentially the innards of a cheap digital watch with a battery backup. When the system is initialized, the contents of the calendar are read into the system clock, which is then maintained separately without referencing the calendar. If the Network Time Protocol (NTP) is in use, the system can be configured to periodically update the calendar in order to correct for its steady drift.

14.2 Get the Current Time

Table 14-1 describes the Cisco IOS functions provided to get the current time from the system clock.

Table 14-1 Functions to Get the Current Time from the System Clock

Time to Get	Function
Current time at a medium resolution (roughly 4 ms)	`vo idclock_get_time(clock_epoch *epoch);`
Current time epoch expressed in microseconds	`ul ongclock_get_microsecs(void);`
Time at the highest resolution supported by the hardware	`vo idclock_get_time_exact(clock_epoch *epoch);`
Current time in UNIX format	`ulong unix_time(void);`
Current time in terms of seconds and nanoseconds since 0000 UTC 1 January 1970	`vo idsecs_and_nsecs_since_jan_1_1970 (secs_and_nsecs *time);`
Current time in ICMP timestamp format (milliseconds since 0000 UTC today)	`ulong clock_icmp_time(void);`

To determine which time protocol set the clock, use the `current_time_source()` function:

```
clock_source current_time_source(void);
```

To determine the current time zone offset, use the `clock_timezone_offset()` function:

```
int clock_timezone_offset(void);
```

To determine the current time zone name, use the `clock_timezone_name()` function:

```
char *clock_timezone_name(void);
```

14.3 Test for Summer Time

It may be useful to test to determine whether a particular epoch falls within the summer time (daylight savings time) period.

To test a clock epoch to determine whether it falls during summer time, use the `clock_time_is_in_summer()` function:

```
boolean clock_time_is_in_summer(clock_epoch *epoch);
```

To test a UNIX-style time value to determine whether it falls during summer time, use the `unix_time_is_in_summer()` function:

```
boolean unix_time_is_in_summer(ulong unix_time);
```

14.4 Convert between Time Formats

Table 14-2 lists the functions available for converting between different time formats.

Table 14-2 Functions for Converting between Different Time Formats

Convert from...	Convert to ...	Function
`clock_epoch`	`timeval`	`vo idclock_epoch_to_timeval(clock_epoch *epoch, timeval *tv, long tz_offset);`
`clock_epoch`	UNIX time value	`ul ong c lock_epoch_to_unix_time(clock_epoch*epoch);`
`timeval`	`clock_epoch`	`vo idclock_timeval_to_epoch(timeval *tv, clock_epoch *epoch);`
`timeval`	UNIX time value	`ul ongclock_timeval_to_unix_time(timeval *tv);`
UNIX time value	`clock_epoch`	`vo idunix_time_to_epoch(ulong unix_time, clock_epoch *epoch);`
UNIX time value	`timeval`	`vo idunix_time_to_timeval(ulong unix_time, timeval *tv, ch ar**tz_name);`

14.5 Set the System Clock

You can set the system clock from either a `clock_epoch` structure or a UNIX-style time value.

To set the clock from a `clock_epoch` structure, use the `clock_set()` function:

```
void clock_set(clock_epoch *epoch, clock_source source);
```

To set the clock from a UNIX-style time value, use the `clock_set_unix()` function:

```
void clock_set_unix(ulong unix_time, clock_source source);
```

14.6 Determine Validity of System Clock Time

An important coding issue is determining whether the time is accurate. If a system is not equipped with a hardware clock or calendar, the system clock will have an unusual value when the system first starts up. Critical time-dependent processes may produce undesirable results if the clock has not been set to an accurate time.

To determine whether the clock has been set, use the `clock_is_probably_valid()` function:

```
boolean clock_is_probably_valid(void);
```

To mark the clock as being accurate, use the `clock_is_now_valid()` function from a time protocol process:

```
void clock_is_now_valid(void);
```

14.7 Format Time Strings

The Cisco IOS provides a rich set of functions for formatting ASCII strings from epochs of various forms.

To get the current time in a fixed format using the local time zone and summer time settings, use the `current_time_string()` function:

```
void current_time_string(char *buf);
```

To format a `timeval` with very flexible formatting options, use the `format_time()` function:

```
ulong format_time(char *buf, ulong buf_len, char *fmt, timeval *time, char *tz);
```

To format a UNIX time value in a fixed format using the local time zone and summer time settings, use the `unix_time_string()` or `unix_time_string_2()` function:

```
void unix_time_string(char *string, ulong unix_time);

void unix_time_string_2(char *string, ulong unix_time);
```

To format a `clock_epoch` with the `printf()` function, use the `%C` format descriptor:

```
printf("%C", fmt_string, epoch);
```

For more information about formatting time strings, see the "Format Time Strings" section in the "Strings and Character Output" chapter

Timer Services

15.1 Overview: Timer Services

The Cisco IOS timer services support two different types of timers. Each type can track time to the limits of the system clock accuracy, which is currently 4 milliseconds. The following are the two timer types:

- Passive timers. Passive timers note the current value of the timer variable `msclock` and then examine this value either by polling periodically or when triggered by an event.
- Managed timers. Managed timers augment passive timers by allowing you to group timers together. This lets you conveniently and efficiently manipulate a large number of timers.

The timer services are used for all time-related functions in the system, such as periodic processes, timeouts, and delay measurements. It is used in all cases where time *intervals* matter. Time-of-day is a separate, but related, function. It is discussed in the "Time-of-Day Services" chapter.

15.1.1 System Clock

Prior to Cisco IOS Release 11.1, timers were based around a 32-bit system clock variable known as `msclock`, which nominally counts the number of milliseconds that have elapsed since the system started up. `msclock` is incremented when a hardware timer fires, invoking the nonmaskable interrupt (NMI) routine. The NMI routine is invoked by the hardware at a nominal interval of 4 milliseconds, so the Cisco IOS code increments `msclock` by four approximately 250 times per second. (Note that, depending on the hardware platform, this rate can range between 248 and 252 ticks per second. This means that `msclock` is an inappropriate mechanism for precision timing, but is otherwise adequate for most uses.) The value of `msclock` is then stored as a timestamp and compared to other timestamps as appropriate.

Because `msclock` is a 32-bit count of milliseconds, it can only count up to about four billion milliseconds, which is just over 49 days, before it rolls over. This effectively means that `msclock` is treated as a circular number space. This implies that durations being timed must be less than one half of the number space (somewhat less than 25 days). It also means that attempts to manipulate timers—particularly, to compare them—are extremely error-prone. For this reason, timers must *never* be manipulated or compared directly; the Cisco IOS timers support *must* be used at all times. Further, the `msclock` variable itself must never be referenced directly. You must use the system support for getting the current time.

Beginning with Cisco IOS Release 11.1, timestamps are now 64 bits wide. This means that they will not roll over for roughly 584,000,000 years, well beyond the mean time between failure (MTBF) of our routers. Applications are no longer allowed manipulate timestamps directly, and the `msclock` variable has been removed from the source code.

Note Use of the `msclock` variable should be avoided at all costs. This variable is present in releases prior to Release 11.1, so referencing it produces code that cannot be ported to all releases.

15.1.2 Implementing Application-Level Functions

The Cisco IOS timer services implement the application-level functions by manipulating timestamps through a set of basic system calls. Typically, when a timer is set to expire at some point in the future, the system calculates the *epoch* (that is, the absolute time) of the expiration, and then the value of the system clock is watched until the expiration epoch is reached.

15.1.3 Timer Jitter

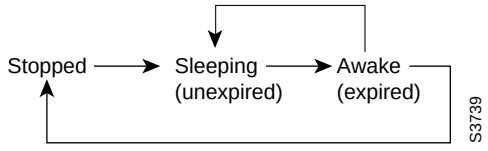
In many applications, it is useful to *jitter* timers. When jitter is applied to a timer, the expiration time is randomized within set limits. It has been observed that periodic router updates tend to become synchronized over time, causing large bursts of routing traffic at regular intervals. The introduction of jitter eliminates this synchronization. Because protocols often have fixed timeout periods, jitter is always subtracted from the time delay, causing the timer to expire somewhat sooner than it was otherwise scheduled to; adding jitter might cause protocol failures. Jitter is available as an option for all timer types.

15.2 Timer States

Passive timers in the future and managed timers can be in one of three possible states (see Figur e15-1):

- Stopped. This is represented by a timestamp of value 0.
- Sleeping (unexpired). A timer that is sleeping is considered to be running.
- Awake (expired). A timer that is awake is considered to be running.

Figure 15-1 Timer States



15.3 Passive Timers

Passive timers (also sometimes called simple timers) take the current value of the system clock and make a note of the value either as it is or after adding a delay value. The as-is time value is used for what are called *timers in the past*. The time with a delay added is used for what are called *timers in the future*. The data structure for a passive timer is a simple timestamp.

Variables declared to hold timestamps must be of type `sys_timestamp`. This variable type is 32 bits in Release 11.0 and earlier, and is 64 bits in Release 11.1 and later.

15.3.1 Passive Timers in the Future

Passive timers in the future, which are the most common timers currently used in the Cisco IOS software, operate around events that are scheduled to occur in the future.

15.3.1.1 Operation of Passive Timers in the Future

Passive timers in the future work as follows:

1. When an event is to be scheduled in the future, the epoch of that event is calculated, typically by adding a delay value to the current epoch. However, there are variations.
2. The Cisco IOS software or the application periodically checks to see if that epoch has been reached, often in a process `BLOCK` routine. When the epoch is reached, the timer is considered "awake" (expired) and the appropriate action is taken.

15.3.1.2 Start a Passive Timer in the Future

When you start a passive timer in the future, you specify a timer and the delay, in milliseconds, after which the timer will expire. You can optionally specify a pseudorandom jitter amount that is a percentage of the delay to subtract from the delay value. With the advent of 64-bit timestamps in Release 11.1, a new function has been added allowing a 64-bit delays to be used when starting a timer. However, starting a jitterred timer is still limited to a 32-bit delta.

To start a passive timer in the future, use the `TIMER_START`, `TIMER_START64`, or `TIMER_START_JITTERED` macro.

```
void TIMER_START(sys_timestamp timer, long delay)

void TIMER_START64(sys_timestamp timer, ulonglong delay)

void TIMER_START_JITTERED(sys_timestamp timer, ulong delay, ulong jitter_percent)
```

The `TIMER_START_ABSOLUTE` and `TIMER_START_ABSOLUTE64` macros start a timer based on the time the router was booted instead of the current time. Use these macros only for timers that should expire at a certain interval after the router is booted.

```
void TIMER_START_ABSOLUTE(sys_timestamp timer, ulong delay)

void TIMER_START_ABSOLUTE64(sys_timestamp timer, ulonglong delay)
```

Once a timer is started, it is considered to be running until it is stopped, even after it has expired. Because timers operate in a circular number space, in releases prior to Release 11.1, if an expired timer is left running for over 24 d ays, it will suddenly look unexpired again. You should always stop or restart timers as appropriate after they expire.

15.3.1.3 Set the Expiration for a Passive Timer

To set a passive timer to expire after a 32-bit or 64-bit delay and adjusted to a boundary interval, use the `TIMER_START_GRANULAR` or `TIMER_START_GRANULAR64` macro, respectively.

```
void TIMER_START_GRANULAR(sys_timestamp timer, ulong delay, ulong granularity)

void TIMER_START_GRANULAR64(sys_timestamp timer, ulonglong delay, ulonglong
granularity)
```

To increase the delay of an existing passive timer in the future, use the `TIMER_UPDATE_GRANULAR` or `TIMER_UPDATE_GRANULAR64` macro.

```
void TIMER_UPDATE_GRANULAR(sys_timestamp timer, long delay, ulong granularity)

void TIMER_UPDATE_GRANULAR64(sys_timestamp timer, ulonglong delay,
                             ulonglong granularity)
```

15.3.1.4 Stop a Passive Timer in the Future

Once a timer has expired, it should be stopped. A stopped timer has a timestamp value of 0. For historical reasons, not all timer macros recognize the value 0. Read the *Cisco Internetwork Operating System API Reference* carefully.

To stop a passive timer in the future, use the `TIMER_STOP` macro.

```
void TIMER_STOP(sys_timestamp timer)
```

15.3.1.5 Determine the State of Passive Timers in the Future

Table 15-1Table 1 5 -1 lists the macros that allow you to determine the state of a passive timer in the future.

Table 15-1 Macros to Determine the State of Passive Timers in the Future

Description	Macro
Determine whether a timer is running (that is, whether it is nonzero).	`bo oleanTIMER_RUNNING(sys_timestamp timer)`
Determine whether a timer is both running (that is, nonzero) and sleeping.	`bo oleanTIMER_RUNNING_AND_SLEEPING(sys_timestamp timer)`
Determine whether a timer is both running (that is, nonzero) and awake.	`bo oleanTIMER_RUNNING_AND_AWAKE(sys_timestamp timer)`
Determine whether a timer has expired.	`bo oleanAWAKE(sys_timestamp timer)` `bo oleanXAWAKE(sys_timestamp timer, ulong maximum_delay)`
Determine whether a timer has not yet expired.	`bo oleanSLEEPING(sys_timestamp timer)` `bo oleanXSLEEPING(sys_timestamp timer, ulong maximum_delay)`
Calculate time left sleeping before a timer expires.	`ul ongTIME_LEFT_SLEEPING(sys_timestamp timer)` `ul onglongTIME_LEFT_SLEEPING64(sys_timestamp timer)`

15.3.1.6 Guidelines for Using the SLEEPING and AWAKE Macros in Releases Prio r to Release11.1

The `SLEEPING` and `AWAKE` macros work properly *only* when it is guaranteed that the expiration time is never more than 24 days in the past or future. This effectively means that timers must be stopped after they expire or set at least once every 24 days, and that they cannot be set to expire more than 24 days in the future. A common bug is to have a very old timer that after 24 days suddenly appears to be running. For example, if a timer has a value of 100 and the current epoch in `msclock` is 0x81000000, the timer looks to be almost 24 days in the future instead of slightly more than 24 days in the past.

15.3.1.7 Guidelines for Using the XSLEEPING and XAWAKE Macros in Releases Prio r to Release11.1

The `XSLEEPING` and `XAWAKE` macros perform functions parallel to those of the `SLEEPING` and `AWAKE` macros. However, they require an additional parameter, which is the maximum duration that the timer can ever use. This value reduces the period of timer ambiguity to the maximum duration of the timer by shifting the sequence space around so that it extends only slightly into the future, but much further into the past. For example, if the maximum duration of a particular timer is 5 seconds, `XSLEEPING` would work properly from just under 49 days in the past until 5seconds in the future. In this case, if a timer has a value of 100 and `msclock` has a value of 0x81000000, the timer correctly looks to be awake rather than sleeping.

`XSLEEPING` and `XAWAKE` add another way of introducing bugs. If you guess incorrectly about the maximum duration of the timer (for example, you think the timer can be set to run for only 5 s econds, but you set it 10 seconds into the future), it will appear to be long expired rather than almost ready to expire. As noted, you should almost never use these macros, because properly maintained timers never exhibit ambiguous behavior.

15.3.1.8 Guidelines for Avoiding Timer Ambiguity

In general, if you call the `TIMER_STOP` macro after a nonrecurring timer expires, and check that a timer is running by calling the `TIMER_RUNNING` macro before calling the `AWAKE` macro (or by calling the `TIMER_RUNNING_AND_AWAKE` macro), you can avoid most of the common pitfalls relating to timer ambiguity.

15.3.1.9 Determine the Earlier of Two Timers

To determine the earlier of two passive timers in the future, use the `TIMER_SOONEST` macro. If one timer is not running, the other is returned. If both timers are not running, a stopped timer (that is, a value of 0) is returned.

```
sys_timestamp TIMER_SOONEST(sys_timestamp timer1, sys_timestamp timer2)
```

15.3.1.10 Compare Passive Timers in the Future

To determine whether two 64-bit timestamps are equal, use the `TIMERS_EQUAL` macro.

```
boolean TIMERS_EQUAL(sys_timestamp timer1, sys_timestamp timer2)
```

To determine whether two 64-bit timestamps are unequal, use the `TIMERS_NOT_EQUAL` macro.

```
boolean TIMERS_NOT_EQUAL(sys_timestamp timer1, sys_timestamp timer2)
```

15.3.1.11 Update Passive Timers in the Future

Under some circumstances, you may want to add a value to the previous expiration epoch of a timer, rather than setting it to the current epoch plus a delay. You might want to do this when a periodic process demands that there be no slip in the time. For example, if you use the `TIMER_START` macro to restart a timer each time it expires, the next expiration may be slightly later than expected because of process latency in the system. Updating a timer rather than restarting it guarantees that the next expiration time is a fixed interval after the previous one. However, one side effect of this method is that the new expiration time might already have passed if the process has been significantly delayed; this causes the timer to expire immediately. This might be what you want if the number of timer expirations needs to reflect the amount of time passed, but it might be undesirable in other circumstances.

To update an existing timer by adding an additional number of milliseconds or by adding an additional number of milliseconds minus a pseudorandom jitter amount that is a percentage of the delay, use the `TIMER_UPDATE` or `TIMER_UPDATE_JITTERED` macro. These macros do nothing if the timer is stopped. With the advent of 64-bit timestamps in Release 11.1, the `TIMER_UPDATE64` macro has been added to allow a 64-bit delays to be used when updating a timestamp. Update with jitter is still limited to 32-bit deltas.

```
void TIMER_UPDATE(sys_timestamp timer, long delay)

void TIMER_UPDATE64(sys_timestamp timer, ulonglong delay)

void TIMER_UPDATE_JITTERED(sys_timestamp timer, long delay, ulong jitter_percent)
```

15.3.1.12 Use One Timer Value to Compute Another

To add a delay to a timestamp in order to create a separate timestamp, use the `TIMER_ADD_DELTA` macro. This macro returns the sum of the timer value plus a delay (delta). This macro does the addition in place on the parameter *time* . Note that `TIMER_ADD_DELTA` works even if the timer is stopped, that is, if the timer value is 0. In some unusual cases, you need to subtract an offset from a future timestamp.You can do this with the `TIMER_SUB_DELTA` macro. With the advent of 64-bit timestamps in Release 11.1, you can add or subtract 64-bit delays to or from a timestamp with the `TIMER_ADD_DELTA64` and `TIMER_SUB_DELTA64` macros.

```
sys_timestamp TIMER_ADD_DELTA(sys_timestamp timer, long delta)

sys_timestamp TIMER_ADD_DELTA64(sys_timestamp timer, longlong delta)

sys_timestamp TIMER_SUB_DELTA(sys_timestamp timer, long delta)

sys_timestamp TIMER_SUB_DELTA64(sys_timestamp timer, longlong delta)
```

15.3.1.13 Example: Passive Timers in the Future

The following example uses passive timers in the future to implement the *Snark* protocol. This protocol requires that an update be sent every `SNARK_UPDATE` milliseconds. The timer is always running. The following are reasonable code fragments for this protocol.

Initialization Routine

```
/*
 * Send the first update.
 */
snark_update(snark_pdb);

/*
 * Or, defer the first update.
 */
TIMER_START(snark_pdb->update_timer, SNARK_UPDATE);
```

snark_update

```
[send update]
TIMER_START(snark_pdb->update_timer, SNARK_UPDATE);
```

snark_block

```
if (AWAKE(snark_pdb->update_timer))
...
```

15.3.2 Passive Timers in the Past

Passive timers in the past are timestamps in which the current time is noted and then the resulting timestamps are periodically examined to see whether enough time has passed for an event to occur. These timers are often used for such purposes as rate-limiting error messages; when a message is emitted, the time that it was emitted is noted. If another request to emit is made, the previously noted time is examined to see if sufficient time has passed before sending the message again. Timestamps for passive timers in the past are always in the past. Therefore, the elapsed time is treated as an unsigned quantity. This means that prior to Release 11.1, an ambiguity results only after 49 days rather than after 24 days as is the case for passive timers in the future. Releases 11.1 and later still have an ambiguity, but it is on the order of 500 million years.

In software prior to Release 11.1, when an event occurs infrequently, perhaps less than one every 49 days, ambiguity can be introduced into passive timers in the past. Therefore, you must guarantee that the event being limited happens at least once e very 49days—although the length of the ambiguity period may be acceptable. This problem does not exist in Releases 11.1 and later, because the timers do not wrap for 500 million years.

15.3.2.1 Determine the Current Time

To note the current epoch in the timestamp, use either the `GET_TIMESTAMP`, `GET_TIMESTAMP32`, or `GET_NONZERO_TIMESTAMP (obsolete)` macro. The `GET_NONZERO_TIMESTAMP` macro does not allow a zero return. This is useful if timers are going to be stopped using `TIMER_STOP` and tested using `TIMER_RUNNING`. With the advent of 64-bit timestamps Release 11.1, the `GET_NONZERO_TIMESTAMP` macro has become obsolete.

```
void GET_TIMESTAMP(sys_timestamp timestamp)

void GET_TIMESTAMP32(ulong timestamp)

void GET_NONZERO_TIMESTAMP(sys_timestamp timestamp)
```

15.3.2.2 Copy a Timestamp

To copy one timestamp into a second one, use the `COPY_TIMESTAMP` macro. Note that in releases earlier than Release 11.1, this macro performs an atomic copy.

```
void COPY_TIMESTAMP(sys_timestamp timestamp1, sys_timestamp timestamp2)
```

In Release 11.1 and later, to atomically copy one timestamp into a second one, use the `COPY_TIMESTAMP_ATOMIC` macro.

```
void COPY_TIMESTAMP_ATOMIC(sys_timestamp timestamp1, sys_timestamp timestamp2)
```

15.3.2.3 Determine the Elapsed Time

To return the amount of time that has elapsed, in milliseconds, since the timestamp, use the `ELAPSED_TIME` or `ELAPSED_TIME64` macro. In software prior to Release 11.1, the result is always an unsigned integer in the range of 0 to 49 days. If the timestamp is more than 49 days old, aliasing results. In Release 11.1 and later, you can use the `ELAPSED_TIME64` function, which returns an integer in the range of 0 to 500 million years.

```
ulong ELAPSED_TIME(sys_timestamp timestamp)

ulonglong ELAPSED_TIME64(sys_timestamp timestamp)
```

15.3.2.4 Determine Whether a Time Is within a Range

To determine whether a time is within or outside of a specified range, use the `CLOCK_IN_INTERVAL` and `CLOCK_OUTSIDE_INTERVAL` macros. Both macros determine whether the current time lies between the timestamp plus some delay. They work for any time interval up to 49 days minus the delay. If just under 49 days have elapsed since the timestamp was noted, these macros return a false positive.

```
boolean CLOCK_IN_INTERVAL(sys_timestamp timestamp, ulong delay)

boolean CLOCK_OUTSIDE_INTERVAL(sys_timestamp timestamp, ulong delay)
```

To determine whether the current time lies within the time bounded by the delay after the router was booted, use the `CLOCK_IN_STARTUP_INTERVAL` macro.

```
boolean CLOCK_IN_STARTUP_INTERVAL(ulong delay)
```

15.3.2.5 Example: Passive Timers in the Past

The following code sample rate-limits a message to no more than once per minute. However, this code fails to emit an error message if an error occurs during the one minute approximately 49 days after the previous time an error occurred.

```
send_error_message:
if (CLOCK_OUTSIDE_INTERVAL(error_time, ONEMIN)) {
    [send message]
    GET_TIMESTAMP(error_time);
}
```

15.3.3 Compare Timestamps

Timestamp comparisons are useful for both timers in the past and timers in the future. However because timestamps are tracked in a circular number space, arithmetic comparisons, such as less than (<) and greater than (>), do not work.

To compare timestamps, use the `TIMER_LATER` and `TIMER_EARLIER` macros. The timestamps must be within 24.8 d ays of each other.

```
boolean TIMER_LATER(sys_timestamp timestamp1, sys_timestamp timestamp2)

boolean TIMER_EARLIER(sys_timestamp timestamp1, sys_timestamp timestamp2)
```

To calculate the time difference between two timestamps, use the `CLOCK_DIFF_UNSIGNED` and `CLOCK_DIFF_SIGNED` macros. If you are working in Release 11.1 or later, you can also use the `CLOCK_DIFF_SIGNED64` and `CLOCK_DIFF_UNSIGNED64` macros. If you are unclear about the time relationship between the two timestamps, use the signed version; it returns a value in the range of –24 days to +24 days (±250 million years for the 64-bit version). If you know *a priori* that the second timestamp is later than the first, use the unsigned version; it returns a value in the range of 0 to +49 days (or 0 to 500 million years for the 64-bit version).

```
ulong CLOCK_DIFF_UNSIGNED(sys_timestamp timestamp1, sys_timestamp timestamp2)

ulonglong CLOCK_DIFF_UNSIGNED64(sys_timestamp timestamp1, sys_timestamp timestamp2)

long CLOCK_DIFF_SIGNED(sys_timestamp timestamp1, sys_timestamp timestamp2)

longlong CLOCK_DIFF_SIGNED64(sys_timestamp timestamp1, sys_timestamp timestamp2)
```

15.4 Managed Timers

15.4.1 Overview: Managed Timers

Managed timers are groups of timers that run together. A *parent* timer is used to represent a group of *leaf* (child) timers. The leaf timers are started and stopped directly, and work similarly to passive timers. The managed timer system maintains the leaf timers in a sorted list and links them all to the parent timer. The parent timer is controlled by the managed timer system and inherits the earliest expiration time of any of the leaf timers. This means that only the parent timer needs to be tested for expiration, which makes it straightforward to determine which timer expired.

15.4.2 Type and Context Values

Each leaf timer carries a type value and an opaque context value.

The *type value* allows the code that processes expired timers to discern one kind of timer from another. This means that multiple timers corresponding to different kinds of events or actions can be linked together.

The *context value* is an opaque 32-bit quantity that can be used for any purpose. It most often carries a pointer to some kind of data structure.

15.4.3 Recursive Managed Timers

You can use the managed timer system recursively. That is, you can hierarchically link several parent timers under yet another parent timer. Using managed timers recursively improves efficiency, because it reduces the cost of the timer sorting operation from $O(N)$ to $O(\log N)$. Recursive managed timers also aid modularity, because each subtree is loosely coupled and can be managed without any direct knowledge of its position relative to other subtrees.

15.4.4 Operation of Managed Timers

Once the timer hierarchy is established, you manipulate leaf timers using start, stop, and update functions. When a leaf timer is started, it is linked into a sorted list attached to its parent. The parent is then set to the earliest expiration time of any of its children. This process is repeated recursively. Thus, the highest parent (called the root) timer always reflects the next timer to expire.

Managed timers can also be manipulated from interrupt routines. If a timer is going to be started, stopped, or updated from an interrupt routine, this fact must be flagged when the managed timer is initialized. The managed timer system automatically propagates this fact in the appropriate places in the timer hierarchy so that interrupt exclusion will be applied when necessary. In general, interrupts are not excluded when manipulating a subtree that does not require exclusion.

Only active (running) timers incur any overhead in the managed timer system. Stopped timers stay out of the way completely

To test whether a timer has expired, you test the root of the tree for expiration. If the root is expired, a single call returns the leaf timer that expired, from which the type and context information stored earlier can be obtained. This leaf timer must then be restarted or stopped, or it will continuously expire.

Starting in Cisco IOS Release 11.0, the scheduler allows a process to be notified when a managed timer expires. This timer is known as a *watched managed timer*. Only one managed timer can be watched. Howeer, because a single managed timer can represent an entire tree of timers, this is not

really a restriction. The scheduler itself supports watched managed timers by linking the watched timer into its own timer tree. This means that the scheduler ultimately has a single managed timer to which every process timer tree is subordinate.

15.4.5 mgd_timer Data Structure

The primary data structure for managed timers is of type `mgd_time` . This should be completely opaque to all callers; code should never look inside of this data structure. This structure is typically embedded directly into another data structure rather than allocated separately and used through a pointer

15.4.6 Guidelines for Using Managed Timers

You need to be to be aware of the following when using managed timers:

- The `mgd_timer` block is 24 bytes in releases prior to Release 11.0 and 28 bytes in Releases 11.1 and later), as compared to 4 bytes for a simple timestamp. For example, if you embed a timer in a data structure of which there are 50,000 copies, this could prove to be a significant amount of overhead.
- The managed timer start and stop functions perform insertion sorts that can be expensive relative to simple timestamps. These calls are not appropriate for something that is updated by the receipt of a data packet, for instance.
- The managed timer functions create webs of pointer linkages. Be careful that any timer that is part of an allocated structure is stopped before freeing that structure, and that no child (leaf) timers are ever used after their parent timer has been freed. It *is* safe to free a structure containing a managed timer if it is first stopped.

15.4.7 Initialize Managed Timers

Managed timers must be initialized before use. The timer hierarchy is determined at initialization time, because each timer's parent is specified while the timer is being initialized.

A parent timer must be initialized before its children. This means that a tree of managed timers must be initialized from the root downward.

Note Managed timers cannot be initialized from interrupt routines.

To initialize a parent timer with its parent timer, use the `mgd_timer_init_parent()` function. To initialize the root timer for an application, specify a parent timer of `NULL`.

```
void mgd_timer_init_parent(mgd_timer *timer, mgd_timer *parent);
```

Note The parent timer must be initialized before attempting to initialize a leaf (child) timer.

To initialize a leaf timer, use the `mgd_timer_init_leaf()` function.You specify the parent timer, and the leaf timer type, and a context pointer. If the timer is to be manipulated from interrupt routines, *`interrupt_environment`* must be set to `TRUE`.

```
void mgd_timer_init_leaf(mgd_timer *timer, mgd_timer *parent, ushort type,
                         void *context, boolean interrupt_environment);
```

If a process is going to be awakened by the expiration of one or more managed timers, notify the scheduler of this fact by calling the `process_watch_mgd_timer()` function and marking the root of the managed timer tree as being watched.

```
void process_watch_mgd_timer(mgd_timer *timer, ENABLESTATE watch);
```

15.4.8 Determine Initialization Status of a Managed Timer

To determine whether a managed timer has been initialized, use the `mgd_timer_initialized()` function.

```
boolean mgd_timer_initialized(mgd_timer *timer);
```

15.4.9 Modify the Timer Type

To set a new type value for a managed timer, use the `mgd_timer_set_type()` function. The timer can be a leaf or parent timer, and must have been previously initialized. Note that this function is the only way to set a type in a parent timer. However, parent timers are normally invisible and do not need types.

```
void mgd_timer_set_type(mgd_timer *timer, ushort type);
```

To return the opaque timer type for a managed timer, call the `mgd_timer_type()` function.

```
ushort mgd_timer_type(mgd_timer *timer);
```

15.4.10 Modify the Timer Context

To set a new context for a managed timer, use the `mgd_timer_set_context()` function. The timer must be a leaf timer and must have been previously initialized.

```
void mgd_timer_set_context(mgd_timer *timer, void *context);
```

To return the opaque timer context for a managed timer, call the `mgd_timer_context()` function. This function can be called for leaf timers only; parent timers do not have a context word.

```
void *mgd_timer_context(mgd_timer *timer);
```

15.4.11 Start a Leaf Timer

To start a leaf timer after a delay from the current time, in milliseconds, or after a delay minus a pseudorandom jitter amount that is a percentage of the delay, use the `mgd_timer_start()` or `mgd_timer_start_jittered()` function. For starting a leaf timer with a 64bit delay, use the

`mgd_timer_start64()` function. These functions can be called regardless of whether the timer is already running, and they can be called from interrupt routines. The timer must have been initialized before calling these functions.

```
void mgd_timer_start(mgd_timer *timer, ulong delay);

void mgd_timer_start_jittered(mgd_timer *timer, ulong delay, ulong jitter_percent);

void mgd_timer_start64(mgd_timer *timer, ulonglong delay);
```

15.4.12 Increase the Delay of a Leaf Timer

To increase the delay of a leaf timer with an additional number of milliseconds or by adding an additional number of milliseconds minus a pseudorandom jitter amount that is a percentage of the delay, use the `mgd_timer_update()` or `mgd_timer_update_jittered()` function. If the timer is stopped, this function does nothing. These functions can be called from interrupt routines. The timer must have been initialized before calling these functions.

```
void mgd_timer_update(mgd_timer *timer, ulong delay);

void mgd_timer_update_jittered(mgd_timer *timer, ulong delay, ulong jitter_percent;
```

15.4.13 Set a Leaf Timer's Expiration

To change a leaf timer's expiration to the value of a timestamp if that value is sooner that the timer's current expiration time, use the `mgd_timer_set_soonest()` function. This function can be used regardless of whether the timer is already running.

```
void mgd_timer_set_soonest(mgd_timer *timer, sys_timestamp timestamp);
```

To set a leaf timer to expire at a specific epoch rather than after a time interval, use the `mgd_timer_set_exptime()` function. This function can be used regardless of whether the timer is already running.

```
void mgd_timer_set_exptime(mgd_timer *timer, sys_timestamp *time);
```

15.4.14 Stop a Managed Timer

To stop a managed timer, use the `mgd_timer_stop()` function. This function can be used for both leaf and parent timers. If the timer is a parent, this function recursively stops all the children of this parent. This is useful for such operations as shutting down a process, because it is not necessary to find all the running timers. This function can be called regardless of whether the timer is already running, and it can be called from interrupt routines. If the timer is not running, this function does nothing.

```
void mgd_timer_stop(mgd_timer *timer);
```

A stopped timer is completely unlinked from the managed timer tree, so it is safe to free the memory containing the timer

Caution If you choose to locate a mgd_timer structure inside of memory that has been allocated using `malloc()`, you must be sure to stop the mgd_timer before freeing the space. Because it is harmless to call `mgd_timer_stop()` on a timer that is already stopped, you should always call `mgd_timer_stop()` on the timer before calling `free()` for the memory area. Failure to do so can cause router crashes with `mgd_timer_set_exptime_internal()` in the backtrace.

15.4.15 Determine the State of a Managed Timer

Table 15-2Table 1 5 -2 describes the functions that allow you to determine the state of a managed timer. These functions can be used for both leaf and parent timers, except as noted, and they can be called from interrupt routines.

Table 15-2 Functions to Determine the State of Managed Timers

Description	Function
Determine whether a timer is running.	`bo oleanmgd_timer_running(mgd_timer *timer)`
Determine whether a timer is both running and sleeping; that is, whether it is unexpired.	`bo oleanmgd_timer_running_and_sleeping(mgd_timer *timer)`
Determine whether a timer is both running and awake; that is, whether it is expired.	`bo oleanmgd_timer_expired(mgd_timer *timer)`
Return the address of the first expired timer in the timer tree.	`leaf _timer = mgd_timer*mgd_timer_first_expired (mgd_timer *parent_timer);`
Return the address of the first running timer in the timer tree.	`leaf _timer = mgd_timer*mgd_timer_first_running (m gd_timer*parent_timer);`
Return the number of milliseconds left before a timer expires.	`lo ngmgd_timer_left_sleeping(mgd_timer *timer);` `lo nglongmgd_timer_left_sleeping64(mgd_timer *timer);`
Return the expiration timestamp for a timer.	`sy s_ti mestampmgd_timer_exp_time(mgd_timer* timer);`

15.4.16 Esoteric Managed Timer Functions

This section discusses some managed timer functions that are used only infrequently. Do not use these functions except in specific cases.

15.4.16.1 Link and Delink Timer Trees

You can link entire timer trees into arbitrary places in other timer trees, and you can cleave off (unlink) a subtree. The only likely user of this is the event-driven scheduler.

Linking and delinking a timer tree are roughly equivalent to starting and stopping a timer, except that the timer, which can be a parent, remains running. In particular, linking a timer tree causes an insertion sort into the parent timer, and delinking stops the parent timer if the delinked timer is the only running leaf timer

To link a timer tree into another timer tree, use the `mgd_timer_link()` function.

```
void mgd_timer_link(mgd_timer *timer, mgd_timer *master, mgd_timer **shadow
                    boolean interrupt_environ);
```

To delink a timer subtree from the rest of the timer tree, use the `mgd_timer_delink()` function. The timer being delinked can be a parent timer and can be running.

```
void mgd_timer_delink(mgd_timer **timer);
```

15.4.16.2 Set Extended Context

Leaf timers carry a single opaque word of context information. Normally, one context word should be enough for a timer. However, managed timers can be declared to have additional context information. Both parent and leaf timers can have this additional context information. Declaring additional context information is the only way to add context information to a parent timer.

To set an extended context for a timer, first declare the timer using the `MGD_TIMER_EXTENDED` function. Do not declare a `mgd_timer` directly.

```
MGD_TIMER_EXTENDED(name, extra_context);
```

Then to set the context words, use the `mgd_timer_set_additional_context()` function.

```
void mgd_timer_set_additional_context(mgd_timer *timer, ulong context_index,
                                      void *context);
```

To retrieve the context value, use the `mgd_timer_additional_context()` function.

```
void *mgd_timer_additional_context(mgd_timer *timer, ulong context_index);
```

15.4.16.3 Create Fenced Timers

Under normal circumstances, code that references timers recursively traverses the tree all the way to the leaf timers, ignoring the intervening parent timers. An example of a function that does this is `mgd_timer_first_expired()`. However, you might want to recursively traverse the tree down to an arbitrary point in the tree only, without going all the way to the leaf timer. To do this, set up a *fence* and marking all timers at a particular level as being *fenced*. Then the *fence-level* timer can be found by recursively traversing down from the master timer until the fence is reached.

To set the fenced state of the timer, use the `mgd_timer_set_fenced()` function.

```
void mgd_timer_set_fenced(mgd_timer *timer, boolean state);
```

To return the first subordinate fenced timer, use the `mgd_timer_first_fenced()` function.

```
mgd_timer *mgd_timer_first_fenced(mgd_timer *timer);
```

15.4.16.4 Convert Timers

A stopped timer can be switched between being a leaf and a parent. This is useful under rare circumstances. The converted timer retains its parent, but type and context information may be lost.

To convert a parent timer to a leaf, use the `mgd_timer_change_to_leaf()` function. Before a parent timer is converted to a leaf, all its children must be stopped, and steps should be taken to ensure that the children will not be started.

```
void mgd_timer_change_to_leaf(mgd_timer *timer);
```

To convert a leaf timer to a parent, use the `mgd_timer_change_to_parent()` function.

```
void mgd_timer_change_to_parent(mgd_timer *timer);
```

15.4.16.5 Traverse a Tree of Managed Timers

Two procedures are provided to aid in traversing (walking) a tree of managed timers. Only running timers are considered to be part of the tree.

To return the next sibling of a timer, use the `mgd_timer_next_running()` function. Because timers are stored in sorted order, the function returns the next member of the subtree that will expire.

```
mgd_timer *mgd_timer_next_running(mgd_timer *timer);
```

To return the immediate child of a timer, use the `mgd_timer_first_child()` function. Use this function to descend to the next level while walking a timer tree.

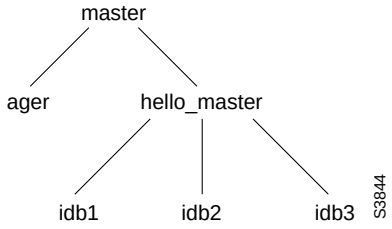
```
mgd_timer *mgd_timer_first_child(mgd_timer *parent_timer);
```

15.4.17 Example: Managed Timers

The following example illustrates the code for a protocol that requires a hello timer for each interface, plus a timer for an ager that runs periodically.

First, decide how to structure the tree. One way is to put the ager timer and all hello timers at the same level under a single master. In addition, suppose you want to display when the next hello will be sent on any interface. Structure the tree to have a single master timer `maste` , under which is the ager timer `ager` and a parent timer `hello_maste` . Under `hello_master` there are individual hello timers for each interface. F igure15-2 illustrates this structure.

Figure 15-2 Sample Managed Timer Tree Structure



The declarations are as follows:

```
mgd_timer master;
mgd_timer ager;
mgd_timer hello_master;
```

The IDB contains the following:

```
mgd_timer idb_hello;
```

Define the timer types as follows:

```
enum {AGER, HELLO};
```

The initialization routine does the following:

```
mgd_timer_init_parent(&master, NULL);
mgd_timer_init_leaf(&ager, &master, AGER, NULL, FALSE);
/*
 *Start the ager.
 */
mgd_timer_start(&ager, 10*ONESEC);
mgd_timer_init_parent(&hello_master, &master);
FOR_ALL_SWIDBS(idb) {
    mgd_timer_init_leaf(&idb->idb_hello, &hello_master, HELLO, idb, FALSE);
    mgd_timer_start_jittered(&idb->idb_hello, 30*ONESEC, 25);
}
process_watch_mgd_timer(&master, ENABLE);
```

At this point, all the timers are running. The protocol handler looks like the following:

```
...
process_wait_for_event(...)
while (process_get_wakeup(&major, &minor)) {
    switch (major) {
    case QUEUE_EVENT:
        /*
         * The queue has packets in it.
         */
        while (...) {
            ...
        }
        break;

    case TIMER_EVENT:
        /*
         * Process all expired timers.
         */
        while (mgd_timer_expired(&master)) {
            mgd_timer *expired_timer;
            idbtype *idb;
            process_may_suspend();
            expired_timer = mgd_timer_first_expired(&master);
            switch (mgd_timer_type(expired_timer)) {
            case AGER:
                run_ager();
                /*
                 * Restart ager.
                 */
                mgd_  timer_update(expired_timer, 10*ONESEC);/* restart ager */
                break;
            case HELLO:
                idb = mgd_timer_context(expired_timer);
                send_hello(idb);
                mgd_timer_start_jittered(expired_timer, 30*ONESEC, 25);
                break;
            default:
                /*
                 * Make it go away!
                 */
                mgd_timer_stop(expired_timer);
                break;
            } /* end switch timer type*/
        } /* end while timer expired */
        break;
    } /* end switch event type */
} /* end while events outstanding */
```

You might have the following display routine:

```
printf("\nNext hello occurs in %d seconds",
       mgd_timer_left_sleeping(&hello_master) / ONESEC);
```

When the process exits, you can stop all the timers as follows:

```
mgd_timer_stop(&master);
```

15.5 Choose Which Type of Timer to Use

Follow these guidelines when deciding which type of timer to use:

- In most cases, use managed timers if you have more than one or two timers. This avoids either having to put many `AWAKE` checks into your block routines or having to run the process periodically to make the `AWAKE` checks.
- Use passive timers for items that are updated at data-forwarding time, because the overhead of a `mgd_timer_start()` can be significant.

15.6 Determine System Uptime

Use the `system_uptime_seconds()` function when you need a measure of time that is monotonically increasing over a long time period. The value returned by this function rolls over after 136 years, so it is effectively guaranteed to always increase and never exhibit aliasing.

```
ulong system_uptime_seconds(void);
```

This function is useful for such operations as timestamping when a link comes up, so that its up time can be displayed and will not roll over after 49 days.

Be careful about the units, however, because this function returns integer seconds, whereas other functions return milliseconds.

Strings and Character Output

This chapter describes the Cisco IOS software functions for printing strings and debugging messages.

16.1 Print Strings

The Cisco IOS software provides several functions that allow you to print strings. Some of these functions are identical in many ways to the ANSI C functions of the same name. However, minor changes have been made to them to support Cisco IOS software-specific needs.

16.1.1 Print a String to the Connected Terminal

To generate output in response to a user command, use the `printf()` function. The output is displayed directly to the currently connected terminal.

```
int printf (const char *format_string, ...);
```

The formatting string consists of text, which is copied verbatim, and format descriptors. Each format descriptor formats one or two parameters from the parameter list, producing an output string. Unlike the standard C `printf()` function, the Cisco IOS `printf()` function allows a single format descriptor to format more than one parameter.

16.1.2 Print a Debugging String

When the platform user enables debugging, the platform prints debugging message on monitor terminals. To format debugging messages, use the `buginf()` function. The formatting strings for this function are the same as those for the `printf()` function.

```
void buginf(const char *format_string, ...);
```

16.1.3 Print a String into a Buffer

To format strings and place them into a buffer, use the `sprintf()` function. The formatting strings for this function are the same as those for the `printf()` function.

```
int sprintf (char *string, const char *format_string, ...);
```

16.1.4 Format Time Strings

The Cisco IOS time-of-day code allows you to format a time string using the systemwide `printf()`, `sprintf()`, and `buginf()` functions, producing a text string from a `clock_epoch` structure. To format a time string, use the `%C` format descriptor for Releases 11.0 and 11.1 and the `%CC` format descriptor for Releases 11.2 and later. Table 16-1 lists the parameters that the `%C` and `%CC` descriptors require. Table 1 6-2 lists the `printf()` modifiers that the `%C` and `%CC` modifiers support.

Table 16-1 Time-String Descriptor Formats

Descriptor Format	Description
`char *`	Format string. See the `format_time()` function in the "Time-of-Day Services" chapter in the *Cisco Internetwork Operating System API Reference* for details about this string.
`clock_epoch *`	Time epoch to convert.

Table 16-2 %C and %CC Descriptor Modifiers

Modifier	Description
nn	Specifies the field width.
`-`	Right-justifies the field.
#	Formats the time as UTC rather than in the local time zone.

16.1.4.1 Examples: Format Time Strings

This section provides several examples of how to format time strings.

In the following code, the `%C` format descriptor indicates that the time string will be formatted from the next *two* parameters passed to `printf()`. The first parameter is a time-formatting string—`%U` means to format the 12-hour time with an AM/PM indication. The second parameter is a time epoch.

```
clock_epoch current_time;
/*
 * Get the current time.
 */
clock_get_time(&current_time);
printf("  The time is %C", "%U", &current_time);/* Use %CC for Releases 11.2 and later.
*/
```

This code displays a string similar to the following:

```
The time is 04:28:57 PM
```

The following example shows how you can combine the standard `printf()` format descriptors with those specific for time strings:

```
clock_epoch current_time;
/*
 * Get the current time.
 */
clock_get_time(&current_time);
printf("At %C, two plus two is %d", "%U", &current_time, 2 + 2);
```

This code displays a string similar to the following:

```
At 04:27:57 PM, two plus two is 4
```

16.1.5 Format Timestamps

To format a timestamp, use the `T` modifier (possible with other modifiers) followed by a single format code. Table 1 6 -3 lists the`printf()` modifiers and Table 16-4 lists the format codes you can use to format timestamps.

When formatting timestamps, in general, the uppercase version of a format code requests a formatted time, and the lowercase letter requests a raw number. Specifying the `l` modifier requests that milliseconds be appended to the number of seconds.

Table 16-3 printf() Modifiers for Timestamps

Modifier	Description
`-`	Right-justifies the field. This modifier is meaningful only if you specify a field width.
`#`	Adds a leading 0 or 0x for octal or hexadecimal formatting codes, respectively.
`l`	Formats integers as long integers instead of as normal integers.

Table 16-4 print() Format Codes for Timestamps

Format Code	Description
`TA` `Ta`	Formats an absolute time, printing the numerical value of a timestamp. The argument is a 64-bit `sys_timestamp`.
`TC` `Tc`	Formats a difference between two timestamps, printing the number of centiseconds. The argument is a 32-bit `ulong`.
`TD` `Td`	Formats a difference between two timestamps, printing the number of milliseconds. The argument is a 4-bit `sys_deltatime`.
`TE` `Te`	Formats an elapsed time, printing the time elapsed since the timestamp. The argument is a 64-bit `sys_timestamp`.
`TF` `Tf`	Formats a future time, printing the time remaining until the timestamp. The argument is a 64-bit `sys_timestamp`.
`TG` `Tg`	Formats a future time of a managed timer, printing the time remaining until the timestamp. The argument is a pointer to a managed timer.
`TM` `Tm`	Formats a difference between two timestamps, printing the number of milliseconds. The argument is a 32-bit `ulong`.
`TN` `Tn`	Formats the current time, printing the numerical value of the current time. This format takes no argument.
`TS` `Ts`	Formats a difference between two timestamps, printing the number of seconds. The argument is a 32-bit `ulong`.

16.1.5.1 Examples: Format Timestamps

Table 16-5 shows examples of formatting the printing of elapsed times. These formats include the following components:

- *ms*—Milliseconds
- *s* or *ss*—Seconds
- *m* or *mm*—Minutes
- *hh* or *y*h—Hours

- *x*d—Days
- *x*w—Weeks
- *x*y—Years

Table 16-5 Examples of Formatting Timestamps

Print Format Specification	Print Output Format
`%TE`	*hh*:*mm*:*ss* *x*d*yy*h *x*w*y*d *xyy*w
`%lTE`	*hh*:*mm*:*ss.mmm* *x*d*yy*h *x*w*y*d *xyy*w
`%#TE`	*m hh*:*mm*:*ss* *x*d*yy*h *x*w*y*d *xyy*w
`%Te`	s
`%lTE`	s.ms
`%#Te`	ms

Table 16-6 shows examples of the output for various time values.

Table 16-6 Examples of Timestamp Output

Value	Time	Output
6000	6 seconds	0
60000	60 minutes	1
600000	10 minutes	10
6000000	1 hour, 40 minutes	1:40:00
60000000	16 hours plus	16:40:00
600000000	6 days plus	6d22h
6000000000	9 weeks plus	9w6d
60000000000	1 year plus	1y47w

16.1.6 Format AppleTalk Addresses

The `printf()` function provides two format codes specific for formatting AppleTalk addresses, `%a` and `%A`.

16.1.6.1 %a Format Code

The `%a` code formats one parameter, a `long`, as either an AppleTalk address, such as 17043.23, or a textual node name, if known, such as *Router.Ethernet1*. All numbers are expressed as decimal.

You can specify optional conversion flags to modify the meaning of `%a`. These are described in Table 16-7.

Table 16-7 printf() %a Conversion Flags

Conversion Flag	Description
`-`	Ignored.
`0`	Ignored.

Conversion Flag	Description
+	Ignored.
#	Converts the value to a textual node name, if known. This is done only if the **appletalk name-lookup-interval** global configuration command is enabled. Otherwise, this flag is ignored.
nn	Minimum field width. Spaces are used to fill the field to this width.
*	Reads the field width from the next parameter.
l	Interprets the parameter to be converted as a 2-byte network number followed by a 1-byte node number and a 1-byte socket number. The socket number is not printed.

If you do not specify the optional `l` flag, the parameter is interpreted as a 3-byte value; the upper 2 bytes are a network number and the lower byte is a node number.

If you do not specify the `#` flag, AppleTalk addresses will appear in one of the following forms:

- *network.node*. This is the default.
- *upper_byte.lower_byte.node*. This form is used only if the **appletalk alternate-addressing** global configuration command is enabled. *upper_byte* is the upper byte of the network number and *lower_byte* is the lower byte of the network number.

Table 16-8 shows examples of using the conversion flags to format the value 0xab0245.

Table 16-8 Examples of Using the printf() %a Conversion Flags

Conversion Flags	Displays Sample Value of 0xab0245 as ...
%a	43778.69
%#a	Router.Ethernet1
%la	171.2

16.1.6.2 %A Format Code

The `%A` code formats one parameter, a `long`, as either an AppleTalk network address or cable range. All numbers are expressed as decimal.

You can specify optional conversion flags to modify the meaning of `%A`. These are described in Table 16-9.

Table 16-9 printf() %A Conversion Flags

Conversion Flag	Description
-	Ignored.
0	Ignored.
+	Ignored.
#	Interprets the parameter to be converted as two network numbers, with the upper 2 bytes as one number and the lower 2 bytes as the other number. Do not specify both the # and l codes.
nn	Minimum field width. Spaces are used to fill the field to this width.
*	Reads the field width from the next parameter.
l	Interprets the parameter to be converted as a 2-byte network number followed by a 1-byte node number and a 1-byte socket number. The socket number is not printed. Do not specify both the # and l codes. This flag is ignored if you specify #.

If you do not specify the optional # or l flag, the parameter is interpreted as a 3-byte value; the upper 2 bytes are a network number and the lower byte is a node number.

If you specify the # flag, AppleTalk cable ranges will appear in one of the following forms:

- *network–network*. This is the default.
- *network* if the lower two bytes are 0. This is the default if the lower two bytes are 0.
- *upper_byte1.lower_byte1–upper_byte2.lower_byte2*. This forms is used only if the **appletalk alternate-addressing** global configuration command is enabled. *upper_byte1* and *upper_byte2* are the upper bytes of the network number, and *lower_byte1* and *lower_byte2* are the lower bytes of the network number.
- *upper_byte.lower_byte.node*. This form is used only if the **appletalk alternate-addressing** global configuration command is enabled and the lower two bytes are 0. *upper_byte* is the upper byte of the network number, and *lower_byte* is the lower byte of the network number.

If you specify the l flag, AppleTalk addresses will appear in one of the following forms:

- *network.node–socket*. This is the default.
- *upper_byte.lower_byte.node–socket*. This form is used only if the **appletalk alternate-addressing** global configuration command is enabled. *upper_byte* is the upper byte of the network number, and *lower_byte* is the lower byte of the network number.

If you do not specify either the # or l flag, AppleTalk addresses will appear in one of following forms:

- *network.node*. This is the default.
- *upper_byte.lower_byte.node* (*network.node*). This form, which prints the address in both formats, is used only if the **appletalk alternate-addressing** global configuration command is enabled. *upper_byte* is the upper byte of the network number, and *lower_byte* is the lower byte of the network number.

Table 16-10 shows examples of using the conversion flags to format the values 0x12345678 and 0x123456.

Table 16-10 Examples of Using the printf() %A Conversion Flags

Conversion Flags	Displays Sample Value of 0x12345678 as ...	Displays Sample Value of 0x123456 as ...
%A	1193046.120[1]	4660.86
%#A	4660-22136	18-13398[1]
%lA	4660.86-120	18.52-86

1 Values printed do not represent meaningful AppleTalk addresses or ranges.

16.1.7 Format Banyan VINES Addresses

The `printf()` function provides two format codes specific for formatting Banyan VINES addresses, `%z` (lowercase z) a n `%Z` (uppercase Z).

16.1.7.1 %z Format Code

The `%z` (lowercase z) code formats two parameters, both `long` types, as a Banyan VINES address. The first parameter is interpreted as a VINES server number and the second is interpreted as a VINES host ID.

You can specify optional conversion flags to modify the meaning of `%z`. These are described in Table 16-11.

Table 16-11 printf() %z Conversion Flags

Conversion Flag	Description
-	Ignored.
0	Ignored.
+	Ignored.
#	Converts the parameters to a textual node name, if known. The value must map from a matching VINES host.
nn	Minimum field width. Spaces are used to fill the field to this width.
*	Reads the field width from the next parameter.
l	Ignored.

If you do not specify the # flag, VINES addresses will appear in one of the following forms:

- *xxxxxxxx*:*xxxx*, where *x* is a hexadecimal digit.
- *uuuuuuuuuu*:*uuuuu*. This format is used only if the **vines decimal** global configuration command is enabled. *u* is a decimal digit.

Server numbers are always padded with zeros to eight digits, or ten digits if the **vines decimal** command is enabled. Host IDs are always padded with zeros to four digits, or five digits if the **vines decimal** command is enabled.

As an example, if you specify the `%z` flag to format the value 0x103030, the value is displayed as 00001030:30 or0000004144:00048 if **vines decimal** is enabled.

16.1.7.2 %Z Format Code

The %Z (uppercase z) code formats one parameter, of type long, as a Banyan VINES server number. You can specify optional conversion flags to modify the meaning of %Z. These are described in Table 16-12.

Table 16-12 printf() %Z Conversion Flags

Conversion Flag	Description
-	Ignored.
0	Ignored.
+	Ignored.
#	Converts the parameter to a textual node name, if known. The value must map from a matching VINES server.
nn	Minimum field width. Spaces are used to fill the field to this width.
*	Reads the field width from the next parameter.
l	Ignored.

If you do not specify the # flag, VINES addresses will appear in one of the following forms:

- *xxxxxxxx*:*xxxx*, where *x* is a hexadecimal digit.
- *uuuuuuuuuu*:*uuuuu*. This format is used only if the **vines decimal** global configuration command is enabled. *u* is a decimal digit.

Server numbers are always padded with zeros to eight digits, or ten digits if the **vines decimal** command is enabled.

As an example, if you specify the %Z flag to format the value 0x103030, the value is displayed as 00103030.

Exception Handling

17.1 Overview: Exception Handling

The Cisco IOS system provides a limited form of exception handling that can be used by processes. This exception handling was originally designed to provide an easy method for processes to catch hardware exceptions, but it has been extended to provide limited software signaling. In no way is the Cisco IOS exception handling intended as a general-purpose signaling mechanism, as there are simple message passing and IPC primitives provided. It is important to note that whenever a process receives a signal, it will be forced from its current point of execution into the signal handler. If the process is currently suspended, it will be scheduled to execute and will begin execution in the handler routine. When the handler routine exits, it will return to the point where the scheduler was called. If the process is executing at the time when the signal is received, it will immediately be forced into the handler routine. When the handler routine exits, the process will continue executing where it was before the signal occurred.

17.2 List of Exceptions

There are a variety of exceptions (or signals) that can occur in the router. The majority of them are related to exceptions in the processor hardware, but several of them are related to the software. Table 17-1 presents the full list of exceptions.

Table 17-1 Exception Signals

Signal	Description
SIGABRT	Used by abort
SIGALRM	Alarm clock
SIGBUS	Bus error
SIGCLD SIGCHLD	Death of a child
SIGSEGV	Segmentation violation
SIGEMT	EMT instruction
SIGEXIT	Sent just prior to process destruction
SIGFPE	Floating-point exception
SIGHUP	Hangup
SIGILL	Illegal instruction

Signal	Description
SIGINT	Interrupt (rubout)
SIGIOT	IOT instruction
SIGKILL	Kill
SIGPIPE	Write on a pipe with no one to read it
SIGPWR	Power-fail restart
SIGQUIT	Quit (ASCII FS)
SIGSYS	Bad argument to system call
SIGTERM	Software termination signal from kill
SIGTRAP	Trace trap
SIGUSR1	User-defined signal 1
SIGUSR2	User-defined signal 2
SIGWDOG	Watchdog timer expiration

17.3 Register an Exception Handler

A process is allowed to register a (possibly different) exception handler for each of the exceptions listed above, with the exception of the SIGKILL exception, which may not be caught. These error handlers can be good for a single use or for as long as the process is executing.

17.3.1 Register a One-Time Handler

A process can register a one-time error handler by calling the signal_oneshot() function:

```
signal_handler signal_oneshot(int signum, signal_handler handler);
```

This function is generally called by a process that expects to produce hardware exceptions.

17.3.1.1 Example: Register a One-Time Handler

As an example, the Cisco 1000 router is designed to trap any bus error caused while accessing its Flash memory. The following code is the exception handler for this:

```
static void c1000_handle_buserror(int signal, int subcode, void *info, char *bad_addr)
{
    longjmp(&berr_buf,1);
}
```

This exception handler simply returns execution to the point before the bus error was generated so that the process can clean up and continue executing.

The following code fragment installs the exception handler. This code fragment installs a one-shot exception handler and then attempts to call the Flash read function. If the read causes a bus error, the exception handler routine will be called, which will return control (via the longjump) to the point just before the error occurred. Note that the code reinstalls the original exception handler before it

finishes. This removes the error handler it installed, which might not have been invoked and therefore might still be active, but it also restores whatever handler might have been present before this routine was called.

```
oh = signal_oneshot(SIGBUS, c1000_handle_buserror);
if (setjmp(&berr_buf) == 0) {
    i = (*devcons->dev_read_wrap_fn)(dev, buf, addr, len);
} else {
    i = 0;
    dev_chk(dev);
}
signal_oneshot(SIGBUS, oh);
```

17.3.2 Register a Permanent Handler

Permanent exception handlers are generally used to catch software errors, as opposed to the one-time handlers that are generally used for hardware errors. A process registers a permanent exception handler by calling the `signal_permanent()` function:

```
signal_handler signal_permanent(int signum, signal_handler handler);
```

This function is generally called by a process that wants to catch a software exception such as the `SIGEXIT` signal that is sent as part of process termination. A handler for this signal can be used to clean up the data structures for a process and release any memory it might be using.

17.3.2.1 Example: Register a Permanent Handler

The following Banyan VINES code registers a handler to clean up when the process is terminated:

```
signal_permanent(SIGEXIT, vines_input_teardown);
```

When the VINES code terminates, the `vines_input_teardown` routine is executed:

```
void vines_input_teardown(int signal, int dummy1, void *dummy2, char *dummy3)
{
    paktype *pak;

    reg_delete_raw_enqueue(LINK_VINES);
    reg_delete_raw_enqueue(LINK_VINES_ECHO);
    process_watch_queue(vinesQ, DISABLE, RECURRING);
    while ((pak = process_dequeue(vinesQ)) != NULL)
        retbuffer(pak);
    delete_watched_queue(&vinesQ);
    vines_pid = 0;
}
```

17.4 Cause Exceptions

Most exceptions are caused by hardware exceptions in the CPU. It is possible to cause software exceptions, but these should be restricted to exceptions caused by the Cisco IOS kernel and sent to processes. A software exception is signaled by calling the `signal_send()` function.

```
void signal_send(pid_t pid, int signum);
```

There should be no need for this routine to ever be caused by a process. Interprocess communication should use one of the other defined methods in the Cisco IOS kernel.

17.4.1 Example: Cause Exceptions

The following example of causing exceptions is the call from the Cisco IOS kernel when a process is killed:

```
/*
 * Give the process one last chance to clean up. If the process is already dead,
 * the code got here as the result of a signal(pid, SIGEXIT) and not a
 * process_kill(pid).
 */
if (!process_already_dead)
    signal_send(forkx->pid, SIGEXIT);
```

Debugging and Error Logging

The Cisco IOS software provides several debugging mechanisms for development engineers and support personnel. These include core file generation, a simple ROM-based debugger, a client debugging stub for host-based debuggers, formatted output routines for high-level tracing, and compile-time options to include additional tracing and logging.

18.1 Debug CPU Exceptions

A *CPU exception* occurs when the executing thread of control attempts to perform an undefined operation, such as accessing an invalid address in memory or dividing by zero. Also, if Cisco IOS software detects an internal error, it executes a CPU-specific instruction to declare a software-detected exception.

When an exception occurs, the Cisco IOS software determines whether the exception can be handled or whether it represents a bug. For example, on some processors, the Cisco IOS software detects misaligned accesses to memory and handles the access in software, returning from the exception. Additionally, some parts of the Cisco IOS software explicitly trap exceptions, for example, when accessing device registers of removable devices.

If the exception is not handled, the router or other platform is automatically reloaded in order to restore the system to a known state. To facilitate debugging after an exception, the Cisco IOS software includes several options for modifying the handling of fatal exceptions.

18.1.1 Use Core Files to Debug CPU Exceptions

A router platform can be configured to generate a core file when a fatal exception occurs. The core file contains an image of all main memory on a platform at the time the exception occurred.

Note Currently, the handling of I/O memory in core files is platform dependent and in many cases is not handled properly.

In Release 12.0, the tools for troubleshooting crashed related to memory corruption were improved. These improvements include the ability to write core files to flash through the **exception flash** command. This command enables / disables writing a core file to flash. While this command is

enabled, if a crash occurs or a command given to write the core file, a core file will be written to flash. The core file on the flash device will be compressed. You can decompress it using gunzip or any unzipper that understands LZ77 coding.

The improvements also include the **memory sanity** command, which is the same as the **debug sanity** command with the enhancement of saving the information in NVRAM, plus a bit more information is saved, such as `caller_pc`, when a buffer has been allocated:

```
[no] memory sanity [trace / queue / chunk / buffer]
```

The **memory sanity** command extends the functionality of debug commands across reboots. In cases of memory leaks, customers often re-boot the router, making it impossible to get core dumps if **debug sanity** was enabled.

A core file is written by the Cisco IOS software using the UDP/IP stack and the regular device drivers. This results in several restrictions with using the core file mechanism:

- A core file can be written only when the thread of control is a process. If the thread of control is the scheduler, a scheduler test predicate, or an interrupt service routine, a core file is not written.
- The process associated with the exception is placed in a special wait condition to keep it from executing further and to save its register context at exception time. If the service of this process is needed for IP to operate (such as the `IP Input` process), a core file cannot be written.
- If the process associated with the exception is receiving a steady source of input packets from the same interface that is used for the core file, the input interface queue might fill, causing the core file write to fail.
- If memory is badly corrupted (such as the packet buffer list being overwritten), further exceptions are likely as the router allocates buffers to write the core file. An exception that occurs while writing a core file aborts the core file writing process.
- Other processes that share data structures with the exception process might experience exceptions if they execute while the core file is being written. These exceptions are treated as fatal errors and abort the core file write.

18.1.1.1 Configure the Cisco IOS Software to Generate a Core File

To generate a core file, perform the following tasks in global configuration mode:

Task		Command
Step	Specify the transfer protocol. If you omit this command, TFTP is used.	**exception protocol** {**ftp** | **rcp** | **tcp**}
Step	If you wish to write the core file to flash, enable this command. If not, disable it with the **no** option. While enabled, this command will write a core file to flash if either a crash occurs or a command to write the core file is given.	[**no**] **exception flash** { **all** / **iomem** / **procmem** } { **dev** [**:***partition*] }
Step	Specify the name of the core file. If you omit this command, the core file is named *routername*-core, where *routername* is the name of the router set with the **hostname** command.	**exception core-file** *dump_filename*
Step	Write the core file.	**exception dump** *dump_host*

18.1.1.2 Analyze a Core File

You can analyze a core file with GDB, grovel, or UNIX tools.

Analyze a Core File with GDB

The Cygnus GNU GDB debugger contains support for reading Cisco IOS core files. For information about using GDB, see the *Cisco EngineeringTools Manual*.

Analyze a Core File with Grovel

You can analyze core files with `grovel`, which is an internal tool for operating on router core files. It is particularly useful for performing operations such as executing the **show ip route** or **show memory** command on a postmortem core dump to examine the state the router was in when it crashed.

Note *grovel* is not for the faint of heart.

You can add functionality to `grovel` by copying the relevant sections of Cisco IOS router code into the grovel framework and using the appropriate macros to correct the addresses.

`grovel` is very dependent upon data structures in the current Cisco IOS router code. *grovel* is maintained manually, so frequently the first step to using `grovel` is to update it to work with the current version of the Cisco IOS code.

To check out a copy of `grovel`, issue the following with your CVSROOT pointing at the appropriate Cisco IOS source repository:

cvs checkout grovel

For more information about *grovel*, contact the Software Development Tools group at the e-mail alias `sw-tools-group`.

Analyze a Core File with UNIX Tools

Core files are simply a dump of memory (although this will change in the future to include header information), so you can use UNIX tools such as *od* can be used to dump the image in hexadecimal for analysis.

18.1.2 Debug with the ROM Monitor

If the four least-significant bits of the configuration register (the boot source specifier) are 0, the system stops at the ROM monitor prompt after an unhandled system exception. The ROM monitor debugger is primitive and is rarely used for normal debugging.

In the ROM monitor, you can enter one of the debugging commands listed inTabl e18-1.

Table 18-1 ROM Monitor Debugging Command

Command	Explanation
alter	Changes and examines memory
stack	Provides a traceback

18.1.3 Debug with GDB

The router contains support for source-level symbolic debugging using the Cygnus GNU GDB debugger. There are two major modes for debugging a router with GDB: kernel mode and process mode. Under normal circumstances, you use GDB kernel mode when debugging Cisco IOS software. In some situations, such as debugging a remote customer's router over the Internet, you must use the more restricted (and dangerous) process mode debugging.

Caution Do not enter any of the commands listed in this section unless you are connected to the router via GDB. Once a debugger command is issued, the router becomes unusable until the host debugger connects to the router.

18.1.3.1 Debug in GDB Kernel Mode

If you have access to the console port of a router, kernel debugging is the preferred way to debug the router. In kernel debugging mode, the entire router is stopped during the exception, freezing all system states.

To enter GDB kernel debugging mode, use the **gdb kernel** EXEC command:

gdb kernel

This command starts the remote debugging protocol and executes a breakpoint. At this point, you can set any breakpoints as needed, or you can continue execution and wait for an exception. Once the **gdb kernel** command has been executed, all unhandled exceptions are passed to the debugging session on the console port.

18.1.3.2 Debug in GDB Process Mode

In some situations, you cannot gain access to the console port of the router. In these situations, you can debug in process mode. Process debugging mode works by intercepting the exceptions of a specified process, placing the process into a special wait state where it will not be scheduled, and then running the process of the debugger to debug the failed process.

Because the Cisco IOS software continues to run during process debugging, it is possible to debug a router over a Telnet session or via a modem connected to the AUX port or, on a communication server, to any port.

There are various restrictions associated with process mode debugging:

1. Only processes can be debugged. Process debugging is not possible for the scheduler, a scheduler test predicate, or an interrupt service routine.
2. The process being debugged is placed in a special wait condition to keep it from executing further and to save its register context at the time of the exception. If the service of this process is needed for the debugging path—such a TCP/IP when debugging over a Telnet session—the process cannot be debugged.
3. If the process being debugged is receiving a steady source of input packets from the same interface as is used for the debugging session, the input interface queue might fill, causing the debugging session to terminate.
4. If memory is badly corrupted—such as the packet buffer list being overwritten—further exceptions are likely as other processes are scheduled. An exception that occurs in any other process is fatal.

5. Other processes that share data structures with the process being debugged can execute in cases where they would not execute before, while the process being debugged is blocked. This can cause exceptions in other processes because of an inconsistent data structure state.
6. Breakpoints in common routines will cause fatal exceptions in other processes that are not being debugged. Single-stepping through a common routine has the same effect, because the debugger inserts breakpoints to implement single-stepping.
7. Exceptions that occur while the process being debugged is running at elevated IPL are fatal and cannot be trapped for process debugging. This includes single-stepping through code that locks out interrupts.

To use process debugging, use the **show process** command to determine the process ID of the process to debug. Then use the **gdb debug** *pid* command, where *pid* is the process ID of the process to debug, to start a debugging session. The next time the process is scheduled for execution, it will execute a breakpoint prior to resuming control.

If you need only read-only access, the **gdb examine** *pid* command is a much safer alternative. It provides read-only access to router memory and the registers of a specified process. It does not block anything or allow write access to memory, so it is difficult to do damage in this mode.

18.2 Debug with buginf() and the debug Command

The router has an extensive collection of internal trace points that you can enable with **debug** EXEC commands. These commands provide formatted output of various internal data structures and trace states for Cisco IOS software components.

It is highly useful to extend the debug mechanism as new features are added to the Cisco IOS software. To do this, place calls to the `buginf()` function at useful places throughout your code. Consider providing the information that you want to see when code is behaving erratically in a platform on which you cannot run GDB.

Calls to `buginf()` are sent through the system logging mechanism, similarly to the more formal `errmsg()` function. This provides logging to multiple terminals and remote `SYSLOG` servers. One consequence of this is that messages can be delayed or lost during periods of heavy logging.

18.2.1 Debug Critical Code Sections

Occasionally, you might need to bypass the system logging mechanism in order to ensure that a message is output. This capability is reserved for critical use only, because it locks out interrupts while running and slows down the system dramatically, ensuring that no messages are lost.

Any such debugging messages should *not* be controlled via the **debug** command. Instead, enable them via compile-time conditionals.

To output a critical message, use the `fprintf()` function with the special destination `CONTT`.

18.3 Debug Using Compile-Time Conditionals

The Cisco IOS software contains several compile-time conditionals to provide additional debugging support. These conditionals often grow data structures or slow down the system, so use them with care.

18.3.1 Trace Buffer Leaks

Various pieces of software occasionally "forget" to return buffers to the free pool when done with them. To get the list of currently allocated buffers, you can use the **show buffer allocated** EXEC command, which prints a list of all allocated buffers, or the **show buffer** *interface* EXEC command, which prints only buffers that sit on the interface input queue for more than 1 minute. If the output of these two commands provides no hint about where the buffers were leaked, you can get more detailed information about the buffer header and buffer data by performing the following tasks:

Task	Command
List the buffer headers.	**show buffers** [**allocated**] [*interface*]
Display the buffer data of selected buffers.	**show memory** *address1 address2*

Another way to determined where the buffers were leaked is to use GDB to print the buffer header and the buffer's data.

However, sometimes all this information is not enough to determine where the leak is occurring. The next step is to find out which routine allocated the leaked buffers. To do this, rebuild the image with a debug flag:

rm buffers.o
make GDB_FLAG="-g -DBUFDEBUG"

Then, issue the **show buffers allocated** command again. The `Alloc PC` field shows the routine that allocated this buffer.

18.3.1.1 Example:Trace Buffer Leaks

The following example traces a buffer leak on Ethernet interface 0.

The **show buffers ethernet 0** command results in output similar to this:

```
Small buffer starting at memory location 0xD2108.

0D2108: 000D1F9C 000D2274 00000000 00005435 ......"t......T5
0D2118: 000056C2 800000A8 00000001 00000000 ..VB...(........
0D2128: 000DB278 00000000 00008003 00000000 ..2x............
0D2138: 00000000 00000014 000D2210 00000000 ..........".....
0D2148: 00000000 00000000 12149484 12149484 ................
        :
Small buffer starting at memory location 0xD23E0.

0D23E0: 000D2274 000D254C 00000000 00005435 .."t..%L......T5
0D23F0: 000056C2 800000A8 00000001 00000000 ..VB...(........
0D2400: 000DB278 00000000 00008003 00000000 ..2x............
0D2410: 00000000 00000014 000D24E8 00000000 ..........$h....
        :
0D2510: 00014BC8 0000012E 00000352 01400007 ..KH.......R.@..
0D2520: 00E20000 000F001F 01C000C3 2E6D4DDA .b.......@.C.mMZ
0D2530: 00000000 00000006 00000003 00000000 ................
0D2540: 00000000 00000000 00000000 000D23E0 ..............#`

Small buffer starting at memory location 0xD254C.
```

You can analyze hex dumps directly or, to make the analysis easier, you can combine the hex output with GDB. Using your favorite text editor, extract the lines that contain the string "starting at" and then prune down to the addresses themselves, for example, (0xd2108), which is a block address. Using GDB—probably the **gdb examine** *process* command so that it does not affect a running system—add in an appropriate "print" and typecasts:

```
print * (paktype *) (((blocktype *)0xd2108)+1)

(gdb) $15 = {next = 0x0, if_input = 0xdb278, if_output = 0x0, flags = 32771, mci
_status = 0, desthost = 0, length = 20, dataptr = 0xd2210 , cb = 0x0, bridgeptr = 0x0,
cacheptr = 0x0, unspecified = {303338628, 303338628}, inputtime = 3033387 28,
datagramsize = 60, enctype = 1, enc_flags = 0, datagramstart = 0xd21ee , lin ktype = 7,
refcount = 1, clns_nexthopaddr = 0x0, clns_dstaddr = 0x0, clns_srcadd r = 0x0,
clns_segpart = 0x0, clns_optpart = 0x0, clns_qos = 0x0, clns_datapart =  0x0,
clns_flags = 0, atalk_srcfqa = 0, atalk_dstfqa = 0, atalk_dstmcast = 0, at alk_datalen
= 0, atalk_dataptr = 0x0, classification = 0 '\000', authority = 0 ' \000', lat_of_link
= 0x0, lat_of_i_o = 0x0, lat_of_data = 0x0, lat_of_size = 0, lat_of_dst = {0, 0, 0},
lat_of_idb = 0x0, lat_groupmask = 0x0, llc2_cb = 0x0, ll c2_sapoffset = 0, llc2_enctype
= 0, llc2_sap = 0 '\000', lack_opcode = 0 '\000',  peer_ptr = 0, e = {encaps = {0
<repeats 17 times>, 255, 65535, 0, 3073, 1289, 2 048, 8200, 49797, 2048}, encapc =
{'\000' <repeats 35 times>, "\377\377\377\000\ 000\f\001\005\t\b\000
\b\302\205\b\000"}}}
```

For example, if the packet is an IP packet, you can add the following print and typecasts:

```
print *(iptype *)(((blocktype *)0xD1EAC)+1)

$69 = {next = 0x0, if_input = 0xdad8c, if_output = 0x0, flags = 32771, mci_statu s = 0,
desthost = 0, length = 20, dataptr = 0xd2564 , cb = 0x0, bridgeptr = 0x0, cacheptr =
0x0, unspecified = {106017556, 106017432}, inputtime = 106017576, da tagramsize = 60,
enctype = 1, enc_flags = 0, datagramstart = 0xd2542 , linktype = 7, refcount = 1,
clns_nexthopaddr = 0x0, clns_dstaddr = 0x0, clns_srcaddr = 0x 0, clns_segpart = 0x0,
clns_optpart = 0x0, clns_qos = 0x0, clns_datapart = 0x0, clns_flags = 0, atalk_srcfqa =
0, atalk_dstfqa = 0, atalk_dstmcast = 0, atalk_da talen = 0, atalk_dataptr = 0x0,
classification = 0 '\000', authority = 0 '\000', lat_of_link = 0x0, lat_of_i_o = 0x0,
lat_of_data = 0x0, lat_of_size = 0, lat_of _dst = {0, 0, 0}, lat_of_idb = 0x0,
lat_groupmask = 0x0, llc2_cb = 0x0, llc2_sap offset = 0, llc2_enctype = 0, llc2_sap = 0
'\000', lack_opcode = 0 '\000', peer_ ptr = 0, e = {encaps = {0 <repeats 19 times>,
519, 260, 13894, 0, 3073, 568, 204 8}, encapc = {'\000' <repeats 38 times>,
"\002\a\001\0046F\000\000\f\001\0028\b\ 000"}}, version = 4, ihl = 5, tos = 0, tl = 40,
id = 33938, ipreserved = 0, dont fragment = 0, morefragments = 0, fo = 0, ttl = 59 ';',
prot = 6 '\006', checksum
```

As another example, to examine TCP data, add the following print and casts:

```
print *(tcptype *) ((iptype *)(((blocktype *)0xD25C8)+1)+1)

(gdb) $70 = {sourceport = 513, destinationport = 995, sequencenumber = 953474086 ,
acknowledgementnumber = 0, dataoffset = 5, reserved = 0, urg = 0, ack = 0, psh = 0, rst
= 1, syn = 0, fin = 0, window = 0, checksum = 33508, urgentpointer = 0 , data =
{"\000\000\000\000"}}
```

Because you can create many of these lines using keyboard macros and then paste them into a GDB window in one operation, you can quickly get a log of all the missing buffers on a system.

PART 4

Network Services

CHAPTER 19

Binary Trees

19.1 Overview: Binary Trees

One common task that must be performed in the router software is storing and retrieving large amounts of information quickly based on a keyed lookup. The Cisco IOS software provides a variety of data structures and utilities in generic libraries that allow you to do this easily. Several data structures and utilities are provided because there are various time and space tradeoffs in choosing a data structure for this task.

Binary trees are suitable for storage and keyed retrieval data structures when the following criteria are present:

- Insertion and deletion manipulations of entries will occur very infrequently relative to the frequency of retrieval.
- The keys for the entries are relatively small, for example, a 32-bit or 64-bit quantity.
- The key space might be relatively sparse or unevenly distributed.
- You will need fast access to any entry in the data structure even when the data structure holds hundreds or thousands of entries.
- The speed of insertions and deletions from the data structure is not as important as the speed of retrieval.

However, binary trees are not without their costs. Binary trees have the following characteristics:

- Insertion and deletion operations might incur high CPU costs as the tree is rebalanced.
- The per-entry memory overhead can be considerable. If each entry you must store is only a few bytes, you should know that the per-entry memory overhead of a binary tree can double or triple your memory usage.
- Binary trees are more complicated than linked lists, hash tables, bags, and arrays.

The Cisco IOS software provides three implementations of binary trees:

- Red-Black (RB) Trees
- AVL Trees
- Radix Trees

19.1.1 Red-Black (RB) Trees

Red-Black (RB) trees are the most general-purpose binary trees in the router library. They are currently used for AppleTalk, VINES, and the OSPF LSA database. The Cisco IOS implementation of RB trees is a threaded tree. That is, once you find a node using a keyed search of the data structure, the only operation necessary to find the next higher or lower node in key order in the data structure is to follow a doubly linked list. RB trees avoid some of the balancing overhead of AVL trees by "coloring" nodes as they are inserted, to postpone the need for balancing and allow the tree to function even when it is slightly out of balance in localized areas of the data structure. Insertions and deletions run in $O(\log n)$ time and searches run in $O(\log n)$ time, but can be made to run in $O(h)$, where h is the height of the tree, in nodes.

A variation on the RB tree—called *interval trees*—is also implemented in the same library as the RB tree. Interval trees are used when the key for an entry has an attribute of *width* or *range*. When the interval tree options are used, the implication is that a key added with its range cannot overlap another key and its range. Think of interval trees as an RB tree with "fat" keys.

19.1.2 AVL Trees

AVL trees are balanced search trees named for Adel'son-Vel'skii and Landis, who introduced this class of balanced search trees. Balance is maintained in an AVL tree by use of rotations; as many as $O(\log n)$ rotations may be required after an insertion to maintain the balance of the tree. In large trees, this may use a considerable amount of CPU time, depending on the tree and the location of the node being inserted. Currently, AVL trees are used in the SSE and IS-IS routing code. (Note, however, that the AVL implementation in the IS-IS routing code is not generic, but rather is specific to IS-IS.) The search time foran AVL tree is $O(\log n)$.

Two levels of AVL functionality are available:

- Raw AVL tree manipulation functions
- Wrapped functions

Raw AVL tree manipulation functions perform the insertion, deletion, balancing and walking of the tree.

Wrapped AVL functions wrap a layer of context around the raw AVL functions. The wrapped functions allow you to insert a node into multipleAVL trees for data that must be sorted on more than one key at once.

19.1.3 Radix Trees

Radix trees are currently used in the IP routing table (both unicast and multicast) and the IP route-cache table. Radix trees lend themselves well toIP, where routing decisions are made by matching not only the route, but also the address mask. The search time for a radix tree is $O(n)$.

19.2 Manipulate RB Trees

19.2.1 Initialize an RB Tree

To allocate and initialize the tree header data structure for an RB tree, use the `RBTreeCreate()` function.

```
rbTree *RBTreeCreate(char *protocol, char *abbrev, char *name,
                     treeKeyPrint printfn, boolean *debug_flag);
```

19.2.2 Insert a Node into an RB Tree

To insert a node into a previously allocated RB tree, use the `RBTreeInsert()` function, which inserts the node in a location based on the specified key structure, or the `RBTreeIntInsert()` function, which inserts the node in a location based on the specified interval.

```
treeLink *RBTreeInsert(treeKey key, rbTree *T, treeLink *node);

treeLink *RBTreeIntInsert(ushort low, ushort high, rbTree *T, treeLink *node);
```

19.2.3 Search an RB Tree

Table 19-1Table 1 9 -1 describes the functions available for searching for nodes in an RB tree.

Table 19-1 Functions for Searching an RB Tree

Search Conditions	Function
Node that exactly matches the specified key value.	`tr eeLink*RBTreeSearch(rbTree *T, treeKey key);`
Overlapping interval in an RB interval tree.	`tr eeLink*RBTreeIntSearch(rbTree *T, t r e e Key);`
First node in the tree.	`tr eeLink*RBTreeFirstNode(rbTree *T);`
Next node in the tree.	`tr eeLink*RBTreeNextNode(treeLink *node);`
Maximal node that is less than or equal to a specified key.	`tr eeLink*RBTreeBestNode(rbTree *T, treeKey key);`
Node equal to or greater than a specified key.	`tr eeLink*RBTreeLexiNode(rbTree *T, treeKey key);`
Node with the largest key.	`tr eeLink*RBTreeNearBestNode(rbTree *T, t r e e Key);`
Node with the largest possible interval key that is less than or equal to the interval specified in the key.	`tr eeLink*RBTreeIntNearBestNode(rbTree *T, treeKey key);`
First node on the tree's internal free list.	`tr eeLink*RBTreeGetFreeNode(rbTree *T);`

19.2.4 Apply a Function to an RB Tree Node

To apply a specified function to each node in the tree in key order, use the `RBTreeForEachNode()` or `RBTreeForEachNodeTilFalse()` function. `RBTreeForEachNode()` applies the function to each node in the tree regardless of what the function returns, and `RBTreeForEachNode()` applies the function until it returns `FALSE`.

```
boolean RBTreeForEachNode(treeProc proc, void *pdata, rbTree *T,

boolean RBTreeForEachNodeTilFalse(treeProc proc, void *pdata, rbTree *T,
                                  treeLink *start, boolean protectIt);
```

19.2.5 Retrieve Information about an RB Tree

Table 19-2Table 1 9 -2 describes the functions available for retrieving information about an RB tree.

Table 19-2 Functions for Retrieving Information about an RB Tree

Information	Function
Number of free nodes that are not busy.	`in tRBReleasedNodeCount(rbTree *T);`
Number of free nodes on the tree's internal free list.	`in tRBFreeNodeCount(rbTree *T);`
Whether a node is on the tree's internal free list.	`bo oleanRBTreeNodeDeleted(rbTree *T, t reeLink*node);`

19.2.6 Print the Nodes in an RB Tree

To format and print all the nodes in an RB tree, use the `RBTreePrint()` function:

```
void RBTreePrint(treeLink *node, ulong depth, rbTree *head);
```

To format and print one node in an RB tree, use the `RBPrintTreeNode()` function:

```
void RBPrintTreeNode(treeLink *node, ulong depth, treeKeyPrint *fn);
```

19.2.7 Protect a Node in an RB Tree

To mark a node in an RB tree as busy, use the `RBTreeNodeProtect()` function. If the node is not busy, it is not deleted if it is passed to `RBTreeDelete()`.

```
boolean RBTreeNodeProtect(treeLink *node, boolean lockIt);
```

To retrieve the protection state of an entry in an RB tree, use the `RBTreeNodeProtected()` function.

```
boolean RBTreeNodeProtected(treeLink *node);
```

19.2.8 Place a Node on the Tree's Internal Free List

To delete a node from an RB tree and place it on the tree's internal free list for possible reuse later, use the `RBTreeDelete()` function.

```
treeLink *RBTreeDelete(rbTree *T, treeLink *node);
```

Note Do not delete a node twice. Mayhem will result.

To collect nodes previously freed with `RBTreeDelete()`, use the `RBTreeTrimFreeList()` function.

```
boolean RBTreeTrimFreeList(rbTree *T);
```

To add a node to the tree's internal free list, use the `RBTreeAddToFreeList()` function. You must manually account for whether the node to be added to the free list is busy and whether it is still linked into the tree.

```
boolean RBTreeAddToFreeList(rbTree *T, treeLink *node);
```

19.2.9 Remove an RB Tree

To deallocate the data structure for an RB tree, including any nodes on the tree, use the `RBTreeDestroy()` function.

```
rbTree *RBTreeDestroy(rbTree *T, boolean *debug_flag);
```

19.3 AVL Trees

The Cisco IOS software provides raw and wrapped AVL functions. The raw AVL functions perform the insertion, deletion, balancing and walking of the tree. These trees are referred to as *AVL trees*. Wrapped AVL functions wrap a layer of context around the raw AVL functions. The wrapped functions allow you to insert a node into multiple AVL trees for data that must be sorted on more

than one key at once. You could think of wrapped AVL trees as having an internal AVL tree implementation, but for almost all purposes, you should think of them as a different implementation of an AVL tree from the raw AVL trees.

Wrapped AVL functions have the following advantages over the raw AVL functions:

- They provide a handle structure—`wavl_handle`—that holds the parameters that the AVL functions need. Using this structure makes bookkeeping easier.
- They allow an item to be threaded onto multiple AVL trees. This is useful for cases where you must search for items sorted on more than one key.

One drawback of AVL trees is that you cannot have multiple elements with the same key value on the tree. This means that, for example, if you want nodes to be threaded onto three trees based on three keys, each of these keys must be unique. For example, you can thread based on IP addresses, but then every entry must have an IP address. You cannot use a special value such as `0.0.0.0` to represent no IP address, because multiple `0.0.0.0` values cannot be threaded onto the IP AVL tree.

When setting up a WAVL tree with the wrapped AVL functions, the first item in the data structure must be an array of `wavl_node_type`, with one element for each desired thread you want. This array contains the information that the wrapped AVL functions need to reference. You also need a `wavl_handle` for every WAVL tree you want to have. You take all actions by passing the handle for the WAVL tree and the `wavl_node_type` for the element you want added or deleted. Note that you cannot manipulate a WAVL tree with the direct AVL functions once the WAVL tree is created. The direct AVL functions do not update the context block used by WAVL trees.

In the comparison functions you register with the `wavl_init()` function and the walker functions you call with the `wavl_walk()` function, you must first call `wavl_normalize()` to readjust the pointer back to the beginning of your data structure.

It is strongly recommended that you provide a front end for all the functions that return a `(void *)` or a `(wavl_node_type *)` with a conversion function that you supply. Doing so allows you to preserve strict typechecking.

19.3.1 Manipulate Raw AVL Trees

19.3.1.1 Initialize an AVL Tree

To initialize an AVL tree, allocate a node of storage of type `avl_node_type`. Then pass a pointer to a `NULL` pointer as the `top` parameter and a pointer to the newly allocated node as the `new` parameter to the `avl_insert()` function. This initializes the newly created node as the "root," or topmost node, in the AVL tree.

19.3.1.2 Insert a Node into an AVL Tree

To insert a node into an AVL tree, use the `avl_insert()` function. This function inserts the node into the AVL tree and rebalances the tree as needed. You must pass in a pointer to a pointer to the tree's top node (this was previously initialized as described in the "Initialize an AVL Tree" section) and a pointer to the node to be inserted (with the key already initialized).

```
avl_node_type avl_insert(avl_node_type ** top, avl_node_type *new,
                         boolean *balancing_needed, avl_compare_type compare_func);
```

19.3.1.3 Traverse an AVL Tree

To traverse (walk) a nAVL tree in lexical order with a function, use the `avl_walk()` function. Think of this as being the functional equivalent of the Lisp *apply* function.

```
boolean avl_walk(avl_node_type *element, avl_walker_type proc, ...);
```

To return the first node in the specified tree (commonly represented as the left node in a conventional drawing of a binary tree), use the `avl_get_first()` function.

```
avl_node_type *avl_get_first(avl_node_type * top);
```

To return the next node in lexical (key) order on the specified thread, use the `avl_get_next()` function.

```
avl_node_type *avl_get_next(avl_node_type * top, avl_node_type element,
                            avl_compare_type compare_func);
```

19.3.1.4 Search an AVL Tree

To search anAVL tree for a specified goal key, use the `avl_search()` function.

```
avl_ node_type *avl_search(avl_node_type* top, a vl_node_type*goal,
                           av l _ c o m p a r e _ f u n c );
```

19.3.1.5 Remove a Node from an AVL Tree

To remove a specified node from an AVL tree, use the `avl_delete()` function. This function rebalances the tree as necessary after the node has been deleted.

```
avl_node_type *avl_delete(avl_node_type ** top, avl_node_type *target,
                          boolean *balancing_needed, avl_compare_type compare_func);
```

19.3.1.6 Free AVL Tree Resources

There are two ways to free resources associated withan AVL tree. One way is to call the `free()` function to free all nodes in the tree at once, without referencing any of the pointers in the tree's AVL node data structure and without passing any of the nodes being deleted to anyAVL tree functions. The second way to freeAVL tree resources, which incurs a higher overhead, is to call `avl_get_first()` and then `avl_delete()` in a loop until `avl_get_first()` returns `NULL`.

19.3.2 Manipulate Wrapped AVL Trees

19.3.2.1 Initialize a Wrapped AVL Tree

To initialize a wrapped AVL tree, use the `wavl_init()` function. In this function, you pass the `wavl_handle` you want to initialize, the number of AVL trees you want under this wrapped AVL tree, and a comparison routing for each of the AVL trees. You must call this function before calling any other wrapped AVL function.

```
boolean wavl_init(wavl_handle * handle, void *(*findblock)(wavl_node_type *), ...);
```

19.3.2.2 Insert a Node into a Wrapped VL Tree

To insert a node into all threads controlled by a wrapper handle, use the `wavl_insert()` function. This function either inserts the node into all threads or into no threads. It does not leave the node inserted into only the threads that were successful if there is any failure to insert into any of the threads.

```
wavl_node_type wavl_insert(const wavl_handle_type *handle, wavl_node_type *node);
```

To insert a node into only one thread of the threads controlled by the wrapper handle, that is, into a specificAVL tree, use the `wavl_insert_thread()` function. Be careful when using this function, because it can leave the set of trees in a strange state.

```
wavl_node_type wavl_insert_thread(const wavl_handle_type *handle, wavl_node_type *node,
                                  int thread);
```

If you are changing only one of the multiple key values, first use the `wavl_insert()` function to delete the node from the specific AVL tree and then use the `wavl_insert_thread()` function to insert with the new key. Deleting from all trees and then reinserting wastes a large number of CPU cycles.

19.3.2.3 Traverse a Wrapped AVL Tree

To traverse (walk) a wrapped AVL tree in the previously initialized context, use the `wavl_walk()` function.

```
boolean wavl_walk(const wavl_handle *handle, int thread, avl_walker_type proc, ...);
```

To return the first node in the specified tree on the specified thread, use the `wavl_get_first()` function.

```
wavl_node_type wavl_get_first(const wavl_handle *handle, int thread);
```

To return the next node in key order on the specified thread, use the `wavl_get_next()` function.

```
wavl_node_type wavl_get_next(const wavl_handle *handle, wavl_node_type *node,
                             int thread);
```

19.3.2.4 Search a WrappedAVL Tree

To search a wrapped AVL tree for a specified goal, use the `wavl_search()` function or the low-level `avl_insert()` function. The `wavl_search()` function calls `avl_search()` with the selected handle and thread number.

```
wavl_node_type *wavl_search(const wavl_handle *handle, wavl_node_type *goal,
                            int thread);
```

19.3.2.5 Remove a Node from a WAVL Tree

To remove the specified node from all threads controlled by the specified handle, use the `wavl_delete()` function.

```
wavl_node_type wavl_delete(const wavl_handle_type *handle, wavl_node_type *node);
```

To delete the node from only one thread of the threads controlled by the wrapper handle, that is into a specificAVL tree, use the `wavl_delete_thread()` function. Be careful when using this function, because it can leave the set of trees in a strange state.

```
wavl_node_type wavl_delete_thread(const wavl_handle_type *handle, int thread,
                                  wavl_node_type *node);
```

19.3.2.6 Reset Pointers

To reset the pointers to the start of the tree structure, use the `wavl_normalize()` function.

```
static inline wavl_node_type * wavl_normalize(avl_node_type * node, int thread);
```

19.3.2.7 Free WAVL Tree Resources

To free any resources associated with a wrapped AVL tree, use the `wavl_finish()` function. It is important to free the resources associated with a tree when you no longer need them. This is because when you create the tree with the `wavl_init()` function, `wavl_init()` calls `malloc()`. If you do not call `wavl_finish()`, a memory leak will result.

```
void wavl_finish(wavl_handle * const handle);
```

19.4 Manipulate Radix Trees

19.4.1 Initialize a Radix Tree

To initialize a radix tree, use the `rn_inithead()` function.

```
int rn_inithead(void **head, int off);
```

19.4.2 Insert a Node into a Radix Tree

To insert a node into a radix tree, use the `rn_addroute()` function.

```
struct radix_node * rn_addroute(void *v_arg, void *n_arg, struct radix_node_head *head,
                                struct radix_node[2] treenodes);
```

19.4.3 Traverse a Radix Tree

Table 19-3Table 1 9 -3 lists the functions available for traversing (walking) a radix tree.

Table 19-3 Functions for Traversing a Radix Tree

Walking Action	Function
Walk a radix tree, calling a function for every node found in the tree.	`in trn_walktree(struct radix_node *rn, rn _ w a l k _ f u n c t i o n function, o..);`
Apply a walking function across the entire tree, locking down any shared data structures the function uses and ensuring a tree node is active before applying the walking function to it. This is inefficient but a good method to use for routines that print to VTYs.	`in trn_walktree_blocking(struct radix_node *rn, rn_walk_function function, ...);`
Apply a walking function across the entire tree, passing arguments to the function, locking down any shared data structures the function uses, and ensuring a tree node is active before applying the walking function to it. This is inefficient but a good method to use for routines that print to VTYs.	`in trn_ walktree_blocking_list(struct radix_node* rn, rn_walk_function function, va_list pointer);`
Walk a radix tree, dismissing control of the processor in the middle of the walk to allow other threads to run.	`in trn_walktree_timed(struct radix_node_head *head, rn_walk_function walker, r n _ s u c c _ f n e x t n o d e, i.o);`

Walking Action	Function
Walk a radix tree, specifying a version key to use to choose the nodes walked in the tree and dismissing control of the processor in the middle of the walk to allow other threads to run.	`in trn_walktree_version(struct radix_node *head, u_ l o ulversion, rn_walk_function function, rn_succ_ver_functio n nextnode, ...);`

19.4.4 Search for a Node in a Radix Tree

To search for a node in the tree, use the `rn_match()` and `rn_lookup()` functions. The `rn_match()` function performs longest-match lookup, and the `rn_lookup()` function searches by address key and mask and requires an exact match.

```
struct radix_node *rn_match(void *v_arg, struct radix_node_head head);

struct radix_node *rn_lookup(void *v_arg, void *m_arg, struct radix_node_head *head);
```

19.4.5 Mark Parent Nodes in a Radix Tree

To mark with a specified version all parent nodes of a specified node up to the root of the tree, use the `rn_mark_parents()` function. This function is used by the `rn_walktree_version()` function.

```
void rn_mark_parents(struct radix_node *rn, u_long version);
```

19.4.6 Delete a Node from a Radix Tree

To delete a node from a radix tree, use the `rn_delete()` function. Make sure the node you are deleting is not referenced in other data structures and itself has no references. There is no internal freelist, so once the node is deleted from the tree, it is up to the caller to manage the storage.

```
struct radix_node * rn_delete(void *v_arg, void *netmask, struct radix_node_head
*head);
```

CHAPTER 20

Queues and Lists

20.1 Overview: Queues and Lists

The Cisco IOS software provides a variety of functions for manipulating linked lists of data structures. These functions fall into two general groups, those for singly linked lists (sometimes also called *queues*) and those for doubly linked lists.

20.1.1 Singly Linked Lists (Queues)

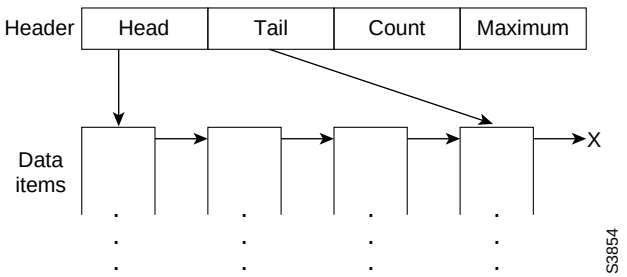
In the original version of the Cisco IOS software, the data structure for singly linked lists was a simple queue in which items were added at the end (tail) of the queue and removed from the beginning (head) of the queue. This simple data structure has developed into a singly linked list structure in which items can be added and removed from any position in the list.

There are two types of singly linked lists:

- Singly linked lists that require that the first longword of the data structure be reserved for linking together the items (also called *direct queues*). Within this category, there are two subsets of functions, those that provide interrupt exclusion and those that do not.

  Figur e20-1 illustrates the relationship between the direct queue data structure header and the actual queue. The “head” field in the header points to the first item at the beginning of the queue, and the “tail” field points to the last item at the end of the queue. The first longword of each data item on the queue points to the next item on the queue, thus chaining together the items in the queue.

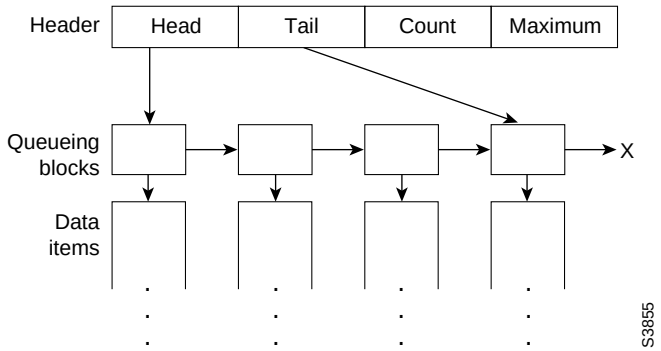
Figure 20-1 Direct Queues



- Singly linked lists with queuing blocks (also called *indirect queues*). These functions have no requirements regarding the format of the data structure. Within this category,the majority of the functions provide interrupt exclusion.

Figur e20-2 illustrates the relationship between the indirect queue data structure header, the queuing blocks, and the actual queue. The “head” field in the indirect queue’s header points to the first in a series of small, intermediary queuing blocks instead of pointing to the first item in the queue. The “tail” field points to the last queuing block. The linkage between the data blocks is in the queuing blocks, not in the data block itself. By using linkages that are not in the data blocks, a data item can be on more than one queue.

Figure 20-2 Indirect Queues



20.1.2 Doubly Linked Lists

The Cisco IOS software provides two types of doubly linked lists:

- Doubly linked lists. The Cisco IOS software provides a few basic functions for adding and removing elements from a doubly linked list. None of these functions provides interrupt exclusion.
- List manager. This is a fully developed set of functions for manipulating doubly linked lists. These functions include code for debugging and for displaying a list and its contents. The Cisco IOS list manager functions place no requirements on the format of the data structure. Using these functions, you can place the same item on multiple data structures. The Cisco IOS functions provide interrupt exclusion on a configurable, per-list basis.

20.2 Manipulate Queues

Most of the Cisco IOS functions for manipulating singly linked lists are specific for direct queues or for singly linked lists with queuing blocks (indirect queues). However, there are a few functions and macros that can be used on all singly linked lists.

20.2.1 Initialize a Queue

To initialize a new queue, use the `queue_init()` function. You can use this function with all singly linked lists, that is, with functions that either have or do not have requirements regarding the format of the data structure.

```
void queue_init(queuetype *queue, int maximum);
```

20.2.2 Determine the State of a Queue

Table 20-1Table 2 0 -1 describes the macros you can use to determine the state of a singly linked list.

Table 20-1 Macros for Determining the State of a Queue

Task	Function
Determine whether a queue is empty.	`bo oleanQUEUEEMPTY(queuetype *queue)`
Determine whether a queue is full.	`bo oleanQUEUEFULL(queuetype *queue)`
Determine whether a queue has a specified amount of space.	`bo oleanQUEUEFULL_RESERVE(queuetype *queue, i n reserve)`
Determine the number of items on a queue.	`in tQUEUESIZE(queuetype *queue)`

20.2.3 Determine Whether an Item Is on a Queue

To determine whether an item is already on a queue, use the `checkqueue()` function. This function does not provide interrupt protection.

```
boolean checkqueue(queuetype *queue, void *data);
```

20.3 Manipulate Direct Queues

20.3.1 Manipulate Unprotected Direct Queues

The functions that manipulate direct queues require that the first longword of the data structure be reserved for linking together the items. Therefore, any data structure that is used with these list functions must be similar to the following:

```
struct xxx_type {
    struct xxx_type *next;
    ...
};
```

This requirement also implies that any item on a direct queue cannot simultaneously be enqueued on another singly linked list.

The functions for unprotected direct queues are identical to the functions for protected direct queues except that they do *not* provide protection from interrupts. Therefore, they cannot be used to pass items from interrupt-level code to process-level code.

20.3.1.1 Add an Item to a Queue

To add an item to the end of a queue, use the `enqueue()` function.

```
void enqueue(queuetype *queue, void *data);
```

To add an item to the beginning of a queue, use the `requeue()` function.

```
void requeue(queuetype *queue, void *data);
```

To insert an item at a relative position in a queue, use the `insqueue()` function.

```
void insqueue(queuetype *queue, void *data, void *previous);
```

20.3.1.2 Remove an Item from a Queue

To remove the first item from the beginning of a queue, use the `dequeue()` function.

```
void *dequeue(queuetype *queue);
```

To remove the next item from an arbitrary point in a queue, use the `remqueue()` function.

```
void *remqueue(queuetype *queue, void data, void *previous);
```

To remove an item from the middle of a direct queue, call the `unqueue()` function.

```
void unqueue(queuetype *queue, void *data);
```

20.3.1.3 Examples: Manipulate Unprotected Direct Queues

This section shows several examples of code of unprotected direct queues.

Example 1

The following example shows how the Cisco IOS Novell IPX code passes packets from one process to another. The IPX input process, which processes packets as they are received from the interfaces, often needs to pass packets to other processes. For example, all IPX Get Nearest Server (GNS) packets are processed by a special IPX process. The two processes perform this packet passing by using a list data structure. The IPX code initializes this queue when it first starts with the following call:

```
queue_init(&novell_gnsQ,0);
```

The IPX input process adds items to this list with the following call:

```
enqueue(&novell_gnsQ, pak);
```

The consumer process removes items from this queue with the following call:

```
pak = dequeue(&novell_gnsQ);
```

Example 2

The following example shows how the IP ICMP code uses the `unqueue()` function. In this example, new echo messages are added to the end of the list and are removed from their current location in the list when the `edisms()` function returns. This code cannot use the `dequeue()` function, because it wants to remove a specific item, which is not guaranteed to be the first item on the list.

```
enqueue(&echoQ, data);
traffic[ICMP_ECHOSENT]++;
edisms((blockproc *)echoBLOCK, (ulong)data);
if (data->active)
    unqueue(&echoQ, data);
```

20.3.2 Manipulate Protected Direct Queues

The functions that manipulate direct queues require that the first longword of the data structure be reserved for linking together the items. Therefore, any data structure that is used with these list functions must be similar to the following:

```
struct xxx_type {
    struct xxx_type *next;
    ...
};
```

This requirement also implies that any item on a direct queue cannot simultaneously be enqueued on another singly linked list.

The functions for protected direct queues are identical to the functions for unprotected direct queues except that they *do* provide protection from interrupts. Therefore, you can use these functions to pass items between interrupt-level and process-level code.

20.3.2.1 Add an Item to a Queue

To add an item to the end of a queue, use the `p_enqueue()` function.

```
void p_enqueue(queuetype *queue, void *data);
```

To add an item to the beginning of a queue, use the `p_requeue()` function.

```
void p_requeue(queuetype *queue, void *data);
```

20.3.2.2 Remove an Item from a Queue

To remove the first item from the beginning of a queue, use the `p_dequeue()` function.

```
void *p_dequeue(queuetype *queue);
```

To remove an item from an arbitrary point in a queue, use the `pak_unqueue()` function.

```
void p_unqueue(queuetype *queue, void *data);
```

To remove the next item after an arbitrary point in a queue, use the `p_unqueuenext()` function.

```
void p_unqueuenext(queuetype *queue, void **previous);
```

20.3.2.3 Example: Manipulate Protected Direct Queues

The following example from the AppleTalk code uses a singly linked list to pass AppleTalk packets from the system drivers running at interrupt level to the process-level code that forwards them, makes routine decisions, and so forth. This queue is initialized with the same function that is used for all singly linked lists.

```
queue_init(&atalkQ, 0);
```

The interrupt-level AppleTalk fragment adds packets to the transfer list by using the following function:

```
p_enqueue(&atalkQ, pak);
```

The AppleTalk input process removes packets from this list by calling the following function:

```
pak = p_dequeue(&atalkQ);
```

20.4 Manipulate Indirect Queue

The functions for singly link lists with queuing blocks (indirect queues) are a derivative of the basic singly linked list functions. Like the basic singly linked list functions, this set of functions does not provide any protection from interrupts. Unlike the basic functions, this set of functions does allow items to be concurrently placed on several linked lists. This means that these functions place no restrictions on the contents of the data structure. These functions maintain a set of small queuing blocks that are used to create and maintain the linkages for the list.

20.4.1 Add an Item to a Queue

To add a packet or an item to the end of an indirect queue, use the `pak_enqueue()` or the `data_enqueue()` function, respectively. These functions provide interrupt protection.

```
paktype *pak_enqueue(queuetype *queue, paktype *pak);

void data_enqueue(queuetype *queue, void *data);
```

To insert a packet at a relative position in an indirect queue, use the `pak_insqueue()` function. This function provides interrupt protection.

```
paktype *pak_insqueue(queuetype *queue, paktype *pak, elementtype *previous);
```

To add an item at any arbitrary position in an indirect queue, use the `data_insertlist()` function. This function does *not* provide interrupt protection.

```
void data_insertlist(queuetype *queue, void *data, void *test_fn);
```

To add a packet to the beginning of an indirect queue, use the `pak_requeue()` function. This function provides interrupt protection.

```
paktype *pak_requeue(queuetype *queue, paktype *pak);
```

20.4.2 Change the Size of a Queue

To change the maximum size of an existing indirect queue, use the `pakqueue_resize()` function. This function provides interrupt protection.

```
void pakqueue_resize(queuetype *queue, int maximum);
```

20.4.3 Iterate over Each Item in a Queue

To iterate over each item in an indirect queue, use the `data_walklist()` function.

```
void data_walklist(queuetype *queue, void *action_fn);
```

20.4.4 Remove an Item from a Queue

To remove the first packet or the first item from the beginning of an indirect queue, use the `pak_dequeue()` or `data_dequeue()` function, respectively. These functions provide interrupt protection.

```
paktype *pak_dequeue(queuetype *queue);

void *data_dequeue(queuetype *queue);
```

To remove a packet from an arbitrary point in an indirect queue, use the `pak_unqueue()` function. This function provides interrupt protection.

```
void pak_unqueue(queuetype *queue, paktype *pak);
```

20.4.5 Examples: Manipulate Indirect Queues

This section shows several examples of the code for indirect queues.

Example 1

The following code fragments are from routines that manipulate the queue of packets waiting to be transmitted on an output interface. Given the likelihood that these packets are also on a retransmission queue somewhere (for example, TCP and LAPB), the output queue manipulation routines must use indirect queues.

The following code fragment, from `holdq_enqueue()`, shows several methods of adding packets—or any item—to a singly linked list variant. This code fragment adds a packet to the beginning or end of a list, depending upon an input parameter:

```
if (which == TAIL) {
    if (pak_enqueue(&(output->outputq[value]), pak)) {
        pak->flags |= PAK_DLQ;
        output->output_qcount++;
        return(TRUE);
    }
} else {
    if (pak_requeue(&(output->outputq[value]), pak)) {
        pak->flags |= PAK_DLQ;
        output->output_qcount++;
        return(TRUE);
    }
}
```

The following code fragment removes a packet from the beginning of the output queue:

```
pak = pak_dequeue(&(idb->outputq[PRIORITY_NORMAL]));
```

Example 2

The TCP code uses these functions to build its retransmission queue. Unlike the `holdq` routines, which work with the head and tail of the list, TCP also adds and deletes its items from arbitrary locations in the list. The following code fragment shows TCP removing an item from its retransmission queue after the item has been acknowledged:

```
pak = pak_unqueue(&tcb->q[RETRANSQUEUE], packet);
```

TCP also sometimes needs to add packets in the middle of its retransmission queue. The following code is used when breaking up a packet into smaller chunks, and all the chunks should be in consecutive positions on the retransmission queue:

```
pak_insqueue(queue, newpaks[i], el);
```

20.5 Manipulate Simple Doubly Linked Lists

The doubly linked list functions provide a fast, straightforward method to build a circular doubly linked list. The following data structure is used to build these lists:

```
typedef struct dqueue_ {
    st  ruct dqueue_*flink;
    st  ruct dqueue_*blink;
    vo            id*parent;
    sy   s_timestampvalue;
} dqueue_t;
```

This data structure can be freestanding, but it is more memory efficient to embed this structure into the items being queued. Multiple instances of this structure can be embedded into the same item, allowing the item to be on many queues at the same time. An additional instance of this structure is also needed to serve as a head/sentinel node for the queue. These functions do not provide any protection from interrupts.

20.5.1 Add an Item to a Doubly Linked List

To add an item at any arbitrary position in a doubly linked list, use the `lw_insert()` function.

```
void lw_insert(dqueue_t *entry, dqueue_t *pred);
```

20.5.2 Remove an Item from a Doubly Linked List

To remove an item from a doubly linked list, call the `lw_remove()` function.

```
void lw_remove(dqueue_t *entry);
```

20.5.3 Example: Manipulate Doubly Linked Lists

The AppleTalk code uses doubly linked list routines fairly extensively. The following example shows how Appletalk enqueues the descriptor of a path to a neighbor device. This code fragment first determines where on the list the new element should be placed and then installs it with the `lw_insert()` function.

```
/*
 * Find appropriate place to insert path. dqhead is a sentinel node.
 */
while ((ndq = dq->flink) != dqhead) {
    path = path_Cast(ndq->parent);
    if (atroute_MetricCompare(&p->metric, &path->metric, ATALK_METRIC_LT))
        break;
    dq = dq->flink;
}
lw_insert(&p->dqLink, dq);
```

The following code fragment removes an item from this doubly linked list:

```
/*
 * Unlink from route's path list.
 */
lw_remove(&p->dqLink);
```

20.6 Manipulate Doubly Linked Lists with the List Manager

20.6.1 Overview: List Manager

The list manager is a fully rounded set of functions for manipulating doubly linked lists. These functions provide a default set of behaviors for manipulating the queues, but allow these behaviors to be modified on a per-queue basis. This allows a process to add an item to the end of one list and insert a new item in sorted order on another list. List linkage and sorting information is maintained within the list structure so that all list accesses are consistent. The list manager also includes code allowing the display of a list and its contents. The implementation of the list needs to supply only a small code fragment to print the contents of a list element. The list manager is responsible for iterating over the list and printing all the list linkage information.

The list manager functions place no requirements on the format of the data structure. Lists linkages are maintained with a small data structure called a `list_element`. This data structure can be embedded into the item being queued, or it can be allocated by the list manager. The use of this extra queueing element allows the same item to be placed on multiple lists with these functions. The list manager also provides interrupt exclusion on a configurable, per-list basis.

20.6.2 Create a List

To create a new list, use the `list_create()` function.

```
list _header *list_create(list_header* list, u s h o maximum, c har* c o n name,
                          us h o flags)
```

20.6.3 Modify an Existing List

When you create a new list with the `list_create()` function, you specify the following flags, which affect the operation of the list:

- `LIST_FLAG_AUTOMATIC` controls whether the list manager should create and delete `list_element` data structures automatically.
- `LIST_FLAG_INTERRUPT_SAFE` controls whether all operations on this list are guaranteed to complete without being interrupted.

Normally, you should not have to change the values of these flags. However, the Cisco IOS software provides two functions that allow you to modify the values in case the list must change its memory allocation paradigm after it has been created.

To change the `LIST_FLAGS_AUTOMATIC` flag on an existing list, use the `list_set_automatic()` function.

```
boolean list_set_automatic(list_header * list, boolean enabled);
```

To change the `LIST_FLAGS_INTERRUPT_SAFE` flag on an existing list, use the `list_set_interrupt_safe()` function.

```
boolean list_set_interrupt_safe(list_header * list, boolean enabled);
```

20.6.4 Add an Item to a List

To add an item to the end of a list, use the `list_enqueue()` function.

```
stat ic inline void *list_enqueue(list_header*list, l ist_element*element, v oid*data);
```

To add an item to the middle of a list, use the `list_insert()` function.

```
stat ic inline void *list_insert(list_header* list, l ist_element*element, v oid*data,
                                 li s t _ i n s e r t _ f u n c t ;u n c _
```

To add an item to the beginning of a list, use the `list_requeue()` function.

```
stat ic inline void *list_requeue(list_header*list, l ist_element*element, v oid*data);
```

20.6.5 Move an Item to Another List

To move an element from one list to another list, use the `list_move()` function.

```
void list_move(list_header *new_list, list_element *element);
```

20.6.6 Remove an Item from a List

To remove the first item from the beginning of a list, use the `list_dequeue()` function.

```
static inline void *list_dequeue(list_header * list);
```

To remove an item from the middle of a list, use the `list_remove()` function.

```
stat ic inline void *list_remove(list_header* list, l ist_element*element, v oid*data);
```

20.6.7 Change the Behavior of List Action Vectors

You can change the default behaviors of the list action vectors called by the `list_dequeue()`, `list_enqueue()`, `list_insert()`, `list_remove()`, and `list_requeue()` wrappers. Tabl e20-2 lists the default behaviors for these wrappers.

Table 20-2 List Action Vector Default Behavior

Function	Default Behavior
`list_dequeue()`	Remove an item from the beginning of the list.
`list_enqueue()`	Add an item to the end of the list.
`list_insert()`	Use the provided function to determine where to add the new item.
`list_remove()`	Remove the specified item.
`list_requeue()`	Add an item to the beginning of the list.

The ability to change the default behavior adds flexibility to how you can manipulate queues, because you can extend or change the default behavior for a list in one place only, without having to propagate the change in many files. For example, to create a stack, you can remap the `list_enqueue()` function to add to the head of the list and then use `list_enqueue()` and `list_dequeue()` to access the stack. (You can also do this with the unmodified `list_requeue()` and `list_dequeue()` functions). If you want a sorted list, you can map the `list_enqueue()` function to a function that inserts the item in sort order. You can also do this with `list_insert()`, but then you must always provide the ordering function. Remapping the vector called by `list_enqueue()` allows you to specify the change once.

The ability to change the functions that the wrappers call allows the characteristics of the physical queue to be changed centrally without changing any of the users of the list. For example, you can change a list to one that is flow controlled based on the amount of data enqueued on it, not merely

on the number of buffers. You can also add callbacks that trap whenever a queue runs dry and have this trigger an event to happen without having to modify and maintain all the code that performs the dequeuing.

To change the behavior of the list modification functions, use the `list_set_action()` function with the list action structure defining the new function vectors to be used.

```
boolean list_set_action(list_header *list, list_action_t *action);
```

20.6.8 Retrieve the Behavior of List Action Vectors

To retrieve the behavior of the list action vectors, use the `list_get_action()` function.

```
list_action_t *list_get_action(list_header *list);
```

20.6.9 Display the Contents of a List

To specify a display function to hold the contents of a list, call the `list_set_info()` function.

```
boolean list_set_info(list_header *list, list_info_t info);
```

To retrieve the function that is called to display the contents of each list item, call the `list_get_info()` function.

```
list_info_t list_get_info(list_header *list);
```

20.6.10 Destroy a List

To delete a list that is no longer needed, use the `list_destroy()` function.

```
static inline void list_destroy(list_header *list);
```

20.6.11 Examples: Manipulate Doubly Linked Lists with the List Manager

This section shows several coding examples of using the Cisco IOS list manager.

Example 1

The following example shows how the new scheduler uses the list manager extensively to keep track of its scheduling lists, lists of events that can wake up a process, lists of events that a particular process is interested in, and so forth.

The scheduler begins by creating all the lists that it needs. The following is an excerpt from this portion of the scheduler code. The final argument to `list_create()`—the `LIST_FLAGS_INTERRUPT_SAFE` flag—indicates that the list manager must provide interrupt protection for all modifications to these queues.

```
/*
 * Initialize the new scheduler lists.
 */
list _create(&procq_ready_c,0, “Sched Critical”, LIST_FLAGS_INTERRUPT_SAFE);
list _c     reate(&procq_ready_h,0, “Sched High”,LIST_FLAGS_INTERRUPT_SAFE);
list _c   reate(&procq_ready_m,0, “Sched Normal”,LIST_FLAGS_INTERRUPT_SAFE);
list _c      reate(&procq_ready_l,0, “Sched Low”,LIST_FLAGS_INTERRUPT_SAFE);
li    st     _create(&procq_idle,0, “Sched Idle”,LIST_FLAGS_INTERRUPT_SAFE);
li    s    t    _    c    r 0, &    a Sched Dead”,LIST_FLAGS_INTERRUPT_SAFE);    q    _    d    e
```

The scheduler then sets up display routines so the scheduler queues can be examined with the list manager's display commands. The following is an excerpt from this portion of the scheduler code:

```
/*
 * Set up the information routines for the basic scheduler lists.
 * The event lists have no information routines available.
 */
list_set_info(&procq_ready_c, process_list_info);
list_set_info(&procq_ready_h, process_list_info);
list_set_info(&procq_ready_m, process_list_info);
list_set_info(&procq_ready_l, process_list_info);
list_set_info(&procq_idle, process_list_info);
list_set_info(&procq_dead, process_list_info);
```

Whenever the scheduler creates a process, it initially always places it on the `procq_idle` list with the following code. The arguments to this function are the new list, the queuing element, and the data structure being added.

```
list_enqueue(&procq_idle, &sp->sched_list, sp);
```

All subsequent manipulations of the process use the `list_move()` function to move the process between the various scheduler lists. The arguments to this function are the new list and the queuing element.

```
list_move(new_list, &p->sched_list);
```

The list on which the item is currently queued is extracted from the queuing element. When a process exits, the `list_remove()` function removes it from the set of scheduler lists. The arguments to this function are the current list, the queuing element, and the data structure being removed.

```
list_remove(&procq_dead, &sp->sched_list, sp);
```

Example 2

The following example shows how the Cisco IOS subsystem code uses the list manager to keep track of all subsystems in the router. It also uses an extra list during subsystem discovery so that it can sequence any subsystems that have prerequisites. This list of pending subsystems is created with the following code. Note that the final argument to this function—the `LIST_FLAGS_AUTOMATIC` flag—indicates that the list manager can dynamically create queuing elements for items added to this list.

```
list_create(&pendinglist, 0, "Subsys Pending", LIST_FLAGS_AUTOMATIC);
```

The subsystem code adds items to the list using the following code fragment. Notice that the `list_enqueue()` call specifies a `NULL` value for the second argument. This signals the list manager to dynamically allocate the queuing element for the entry.

```
/*
 * If the subsystem has a sequence property, it goes on the pending queue.
 */
if (subsys_get_property_list(subsys, subsys_property_seq, NULL))
    list = &pendinglist;
list_enqueue(list, NULL, subsys);
```

After the subsystem code has discovered all subsystems running of a particular type, it runs the pending queue to start the subsystems that had prerequisites:

```
/*
 * For all the subsystems in the pending list, dequeue each one in turn, and
 * evaluate whether their sequence properties have been met.
 */
subsys = list_dequeue(&pendinglist);
while (subsys) {
    /*
     * Attempt to sequence it.
     */
    subsys_sequenced_insert(subsys, property_chunk, 0);

    /*
     * Grab the next victim.
     */
    subsys = list_dequeue(&pendinglist);
}
```

Example 3

The following example from the scheduler code shows the destruction of a list—specifically a watched variable—that is no longer needed. The scheduler has a list of "wakeup" blocks that are threaded onto two lists, one by event and one by process. All these blocks on the event list need to be deleted. The code fragment shows the event being removed from the master list for its event type, all the wakeup blocks for this event being deleted, and then the event-specific list being deleted:

```
/*
 * Remove from the master list for this class of event.
 */
list_remove(event->by_class.list, &event->by_class, event);

/*
 * Free all wakeup blocks that are attached to this event. Using list_dequeue
 * cleans up the 'wi_by_event' thread, so only the 'wi_by_process' thread is left.
 */
while ((wakeup = list_dequeue(&event->wakeup_list)) != NULL) {
```

```
        list_remove(wakeup->wi_by_process.list, &wakeup->wi_by_process, wakeup);
        free(wakeup);
    }
    list_destroy(&event->wakeup_list);
```

Switching

This chapter discusses some of the routing and switching designs in the Cisco IOS code.

21.1 Overview: Switching

The following four different classes of switching can occur within a Cisco router. How many and which of these classes are present depends upon the particular router and its hardware configuration.

- Slow Switching (also known as routing). This class of switching is present in all routers.
- Fast Switching. This class of switching is present in all routers.
- Autonomous Switching. This class of switching is present only in routers with a ciscoBus, CxBus, or CyBus controller.
- Silicon Switching. This class of switching is present only in routers with an SSE card.

21.1.1 Slow Switching

Slow switching, which is generally known as routing, always occurs at process level in the router. Slow switching involves the building of routing tables and forwarding of packets. These tasks are divided into three processes:

1. The first process removes packets from an input queue where they were placed by an interface driver, performs a routing lookup for each packet, and either queues the packet for transmission on another interface or queues it for the second process. This first process is always named after its protocol and function (for example, IP Input), and it always runs at high priority
2. The second process is responsible for processing all packets destined for the router itself, except for routing updates. The routing updates for that network layer protocol are enqueued for a separate routing process. This second process is always named after its protocol and function (for example, AT [AppleTalk] Background), and it always runs at medium priority.
3. The third process is normally where routing updates are processed and routing tables are maintained. This process is always named after its protocol and function (for example, VINES Router), and it always runs at medium priority.

   More than one routing process for a protocol can be active at the same time. For example, a single router might be running BGP, IPEnhanced IGRP, andIP RIP—all IP routing protocols—at the same time.

21.1.2 Fast Switching

Fast switching is a method of performing a routing lookup and forwarding a packet from interrupt level. This technique avoids queueing the packet for a process, the latency of scheduling the process, and any latency within the process itself. Fast switching depends on a special lookup table that is maintained in processor memory by the individual routing processes. Fast-switching code can become very complex because it contains minute details about each type of interface that it supports. The performance of fast-switching code is very critical because it runs at interrupt level.

21.1.3 Autonomous Switching

Autonomous switching operates only on ciscoBus, CxBus, and CyBus controllers. It is a method of performing a routing lookup and forwarding a packet from the controller card without interrupting the main CPU. This technique avoids all the delays that fast switching avoids, and it further avoids the latency of copying the packet across the backplane and any latency in the main processor's interrupt path. Autonomous switching depends on a special lookup table that is maintained on the controller card by the individual routing processes. Autonomous-switching code is very complex because it contains minute details about each type of interface that it supports.

21.1.4 Silicon Switching

Little is known about the silicon-switching code. It is a very reclusive creature that comes out only to look at the light of a blue moon. It is, however, reported to be faster than a bat out of hell, but no one knows for sure.

21.2 Fast Switching

There are two issues to consider when writing fast-switching code. The first is the hardware architecture to which you are writing, and the second is the style of connecting input interface routines to output interface routines. The two issues are orthogonal, so they are discussed separately.

21.2.1 Hardware Architecture

Fast-switching code depends on the hardware architecture. Cisco platforms use one of the following hardware architectures:

- MCI/CiscoBus Architecture
- Shared-Memory Architecture

21.2.1.1 MCI/CiscoBus Architecture

The original Cisco routers were built around a Multibus backplane, and contained third-party interface cards that used Multibus I/O space for passing commands and Multibus memory space for passing data. These cards were all superseded by higher-density Cisco-designed cards that used the Multibus I/O space for passing both commands and data. The discussion in this section applies mainly to these newer cards, although it also applies to the CSC-1R and CSC-2R shared-memory cards. The principal cards in this class are the MCI, the FSIP, and the ciscoBus controller card. Through the ciscoBus controller, this type of fast-switching code can also access other cards, including the EIP, HIP, SIP, and TRIP

This type of fast switching is frequently referred to as “high-end” or “hes” fast switching, although this is a misnomer because the Cisco 7500 series uses the shared-memory model for fast switching.

Receive a Packet

The ciscoBus and MCI cards preclassify a packet, so that when the processor card is interrupted with a packet, the protocol contained in that packet is already known. Given that information, the interface driver can quickly pass a received packet on to the appropriate protocol-specific fast-switching routine.

The ciscoBus/MCI fast-switching routines are almost always specific to a particular encapsulation of a given protocol. For example, there is an AppleTalk ARPA switching routine, an AppleTalk SNAP Ethernet switching routine, and so on. There are a few instances where a fast-switching routine might need to doublecheck the encapsulation type because the ciscoBus or MCI might classify more than one type of packet under the same ID code. An example is the classification of a packet containing a VINES ARPA encapsulation, where the received packet must be explicitly checked to see whether it is actuallyan ARPA-encapsulated packet or instead is a misclassified SNAP-encapsulated packet. Later versions of MCI microcode correctly indicate the difference between these two VINES encapsulations on Ethernet.

When a ciscoBus fast-switching routine receives a packet, it is passed a single argument, a pointer to the input hardware interface. The fast-switching routine is responsible for extracting the necessary information from the interface card, doing this by sending a series of commands to the interface card. The typical sequence of commands does the following:

1. Set the “read pointer” to a particular offset within the ciscoBus buffer.
2. Read enough consecutive words to be able to process the packet. In most protocols, this involves reading the destination address, a flags word, and sometimes some additional data. This data is stored either in registers or in per-IDB variables so that it can be referenced multiple times without having to reread the data across the bus.

The following example, taken from the VINES code, illustrates a typical sequence of commands. This example assumes that the packet is a SNAP-encapsulated Ethernet packet. Lines 1 and 2 set the read offset, and all subsequent accesses must be in `shortwords` or `longwords`. `longword` accesses are preferable whenever possible, because they require fewer CPU cycles to read the same amount of data.

```
/*
 * Set starting location to read from.
 */
inreg->argreg = MCI_ETHER_OFFSET + E_SNAP_HDR_WORDS_IN;
inreg->cmdreg = MCI_CMD_RX_SELECT;

/*
 * Suck in the data.
 */
input->checksum_length = inreg->readlong;
input->hops_ptype = inreg->readshort;
input->destination_net = inreg->readlong;
input->destination_host = inreg->readshort;
```

The following code fragment, taken from the AppleTalk Ethernet ARPA fast-switching code, shows how to reference a byte if necessary when making the fast-switching decision.

```
ch arlongsniff1, sniff2;

srcreg->argreg = AT_ETALK_OFFSET;
srcreg->cmdreg = MCI_CMD_RX_SELECT;
sniff1.d.lword = srcreg->readlong;
if (sniff1.d.byte[0] != ALAP_DDP_LONG)
    return (FALSE);
input->hop_len_word = (sniff1.d.byte[1] << 8) | sniff1.d.byte[2];
sniff2.d.lword = srcreg->readlong;
input->dst_net = (sniff2.d.byte[1] << 8) | sniff2.d.byte[2];
sniff1.d.lword = srcreg->readlong;
input->src_net = (sniff2.d.byte[3] << 8) | sniff1.d.byte[0];
input->dst_node = sniff1.d.byte[1];
input->src_node = sniff1.d.byte[2];
input->dst_sock = sniff1.d.byte[3];
```

Make the Forwarding Decision

Once the fast-switching routine has read the destination address and any other necessary data, it must determine whether the packet should be forwarded. This is done principally by looking up the destination address in a special cache, but it might also involve operations such as checking for the presence or absence of certain bits in a flags word.

If the packet is to be fast switched, the cache entry contains a pointer to the output interface and the encapsulation header to be used on that interface. The correct output routine is called using one of the two methods described in the section "Software Architecture."

If a cache entry is not found or any of the other tests fail, the packet cannot be fast switched and must be handed over to process level. To do this, the fast-switching routine returns `FALSE`. The drivers then ensure that the packet is sent to process level.

Transmit a Packet

The fast-switching output routine is responsible for modifying the received packet and transmitting it. This routine generally has a pointer to the input interface (remember the packet is still on the MCI card or ciscoBus controller) and a pointer to the cache entry. The output routine must first determine whether the input and output interfaces are on the same card or whether the packet must be copied from one interface card to another. Note that all ciscoBus interfaces are considered to be on the same "card," the ciscoBus controller. This decision involves checking whether a card-to-card transfer—such as a ciscoBus-to-FSIP, ciscoBus-to-MCI, or MCI-to-MCI transfer—is necessary.

Transmit a Packet: Intracard

If the two interfaces are on the same card, the fast-switching output routine can simply do the following:

1. Move the packet from the input interface's receive queue to the output interface's transmit queue.
2. Rewrite the packet encapsulation.
3. Tell the controller the new packet starting location and length.

The following example shows how this is done in the VINES code. Note that the controller card can be accessed only in `shortword` or `longword` references, which is the same as in the input routine. Also, the last starting location set for writing data to the packet is the starting location for transmitting the packet. This means that for a protocol that has both a header and a trailer, such as

SMDS, the trailer must be written first. If this rule is not followed, the controller will transmit the proper number of bytes, but it will begin transmitting with the first byte of the trailer instead of the first byte of the header.

```
/*
 * First, set new starting location and length.
 */
inreg->argreg = input->fast_net_start - E_ARPA_HDR_WORDS_OUT;
inreg->cmdreg = MCI_CMD_TX1_SELECT;
size_to_xmit = size_of_data + path->reallength;

/*
 * Write the new header.
 */
inreg->write1long = path->vinesp_mh.mac_longs[0];
inreg->write1long = path->vinesp_mh.mac_longs[1];
inreg->write1long = path->vinesp_mh.mac_longs[2];
inreg->write1short = TYPE_VINES;
inreg->write1long  = input->checksum_length;
inreg->write1short = input->hops_ptype;

/*
 * Now send the packet.
 */
if (size_to_xmit < MINETHERBYTES)
    size_to_xmit = MINETHERBYTES;
inreg->argreg = size_to_xmit;
inreg->cmdreg = MCI_CMD_TX1_START;
```

The following example illustrates the restriction of `shortword` or `longword` accesses to the controller. An FDDI header is nominally 21 bytes in length if you ignore the RIF fields (as this example does). The problem is how to write a 21-byte header that ends on an even byte boundary to a device that can only accept an even number of bytes. The solution is to write a garbage starting byte before the real header, rounding out the total number of bytes written to an even number. In this case, `FDDI_LLC_FC_BYTE` is a constant that is written into the first `shortword` but only the low-order byte is significant. Once you have written the odd-length header, you must signal the controller to ignore the garbage byte when transmitting the packet. This is done at the same time the packet length is sent to the controller by adding the constant `MCI_TX_ODDALIGN` to the packet size. Because the first byte written into memory was a garbage byte, this means that the packet transmission will begin with

the first real byte of the packet. This procedure is necessary only for encapsulations that have odd lengths. This includes FDDI encapsulation, raw 802.5 Token Ring encapsulations, and raw Ethernet 802.3 encapsulations.

```
/*
 * First, set new starting location and length.
 */
inreg->argreg = input->fast_net_start - F_SNAP_HDR_WORDS_OUT;
inreg->cmdreg = MCI_CMD_TX1_SELECT;
size_to_xmit = size_of_data + path->reallength;

/*
 * Write the new header.
 */
inreg->write1short = FDDI_LLC_FC_BYTE;
inreg->write1long = path->vinesp_mh.mac_longs[0];
inreg->write1long = path->vinesp_mh.mac_longs[1];
inreg->write1long = path->vinesp_mh.mac_longs[2];
inreg->write1long = (SNAPSNAP << 16) | (LLC1_UI << 8);
inreg->write1long = TYPE_VINES2;
inreg->write1long = input->checksum_length;
inreg->write1short = input->hops_ptype;

/*
 * Now send the packet.
 */
inreg->argreg = sixe_to_xmit | MCI_TX_ODDALIGN;
inreg->cmdreg = MCI_CMD_TX1_START;
return(TRUE);
```

Transmit a Packet: Intercard

If the two interfaces are not on the same card, the output routine must do the following:

1. Allocate a buffer on the output interface card.
2. Write the new encapsulation header.
3. Copy the contents of the packet from the input interface card to the output interface card.
4. Tell the output controller the packet starting location and length.

The following example shows how this is done in the VINES code. The result of a `MCI_CMD_TX1_RESERVE` command must be checked for every packet. This command does not complete immediately, so interface accounting code (not shown in the example) is usually inserted between the code that issues the command and checks its result. If a new buffer cannot be obtained for output, there are different code execution paths based upon whether priority queuing is in use. With priority queuing, all packets that cannot be set immediately must be bumped up to process level so they can be sorted into the appropriate queues. If priority queuing is off, bumping the packet to process level is considered a waste of time, so it is dropped immediately. Always remember to include a `MCI_CMD_RX_FLUSH` command when you are done with the packet. If you forget, the controller considers that this packet is still being processed, and when the driver requests the next

packet, the controller generates an error message. This command is not necessary in the intracard case, because the original packet has been moved to a transmit queue and is no longer at the head of the receive queue.

```
/*
 * Acquire a buffer on the output interface.
 */
outreg->argreg = output->mci_index;
outreg->cmdreg = MCI_CMD_SELECT;
outreg->argreg = output->buffer_pool;
outreg->cmdreg = MCI_CMD_TX1_RESERVE;
if (outreg->cmdreg != MCI_RSP_OKAY) {
    if (output->priority_list) {
        /*
         * If sorting traffic and interface is congested, process switch.
         */
        return(FALSE);
    } else {
        /*
         * Reserve failed on output. Flush the packet.
         */
        output->outputdrops++;
        inreg->cmdreg = MCI_CMD_RX_FLUSH;
        return(TRUE);
    }
}
```

Because the intercard code is writing to a new packet, it can simply start writing the encapsulation header at the beginning of the buffer. This is different from the intracard case where the new encapsulation header must be overlaid upon the previous encapsulation of a packet.

```
/*
 * Set up the write pointer, and write the new header.
 */
outreg->argreg = 0;
outreg->cmdreg = MCI_CMD_TX1_SELECT;
outreg->write1long = path->vinesp_mh.mac_longs[0];
outreg->write1long = path->vinesp_mh.mac_longs[1];
outreg->write1long = path->vinesp_mh.mac_longs[2];
outreg->write1short = TYPE_VINES;
outreg->write1long = input->checksum_length;
outreg->write1short = input->hops_ptype;
outreg->write1long = input->destination_net;
outreg->write1short = input->destination_host;
```

The following call to `mci2mci()` copies the remaining contents of the packet from the input interface card to the output interface card. It is dependent on the read pointer being at a known position (that is, where it was left by the input fast-switching routine). This example explicitly writes the first 12 bytes of the network layer header before calling `mci2mci()`, because the code had already read past these bytes during the input routine, and it is quicker to write them directly than to reset the read pointer and let `mci2mci()` copy them.

```
/*
 * Copy the remainder of the packet.
 */
mci2mci(&inreg->readlong, &outreg->write1long, size_to_copy);
```

Then, set the length of the packet and transmit it. The starting location of the packet was set at the beginning of this example and never changed, even though numerous writes were performed. Always remember to flush the original packet once it has been copied and transmitted.

```
/*
 * Send the packet.
 */
if (size_to_xmit < MINETHERSIZE)
    size_to_xmit = MINETHERSIZE;
outreg->argreg = size_to_xmit;
outreg->cmdreg = MCI_CMD_TX1_START;

/*
 * Flush the original packet.
 */
inreg->cmdreg = MCI_CMD_RX_FLUSH;
return(TRUE);
```

21.2.1.2 Shared-Memory Architecture

Fast switching on the more recent routers is designed around a shared-memory model. These routers include the Cisco 1000, Cisco 2500, Cisco 4000, Cisco 4500, and Cisco 7500 series. Through the Integrated Route Switch Processor, shared-memory fast-switching code can also access other processors including the EIP,HIP,SIP,and TRIP.

Shared-memory fast switching is frequently referred to as *low-end* or *les* fast switching, although this is a misnomer because the Cisco 7500 is at the top of the router continuum.

Receive a Packet

On the shared-memory platforms, the interface drivers classify the packets. With this information, the interface driver can quickly pass a received packet on to the appropriate protocol-specific fast-switching routine. The shared-memory fast-switching routines are always specific to a particular encapsulation of a given protocol. For example, there is an AppleTalk ARPA switching routine, an AppleTalk SNAP Ethernet switching routine, and so on.

When a shared-memory fast-switching routine receives a packet, it is passed a single argument, which is a pointer to the packet data structure. This data structure contains a pointer to the input interface that the routine can use for access checks and other operations. The contents of the packet are completely accessible to the routine simply by referencing through the packet data pointer. It is the responsibility of the fast-switching routine to extract the information from the packet necessary to be able to process it. In most protocols, this involves reading the destination address, a flags word, and sometimes some additional data. This data is stored either in registers or in per-IDB variables so that it can be referenced multiple times without having to access the data again from slow shared memory.

The following is a typical sequence of instructions for a shared-memory fast-switching routine that receives a packet. This example is from the VINES code. This code does not care which interface the packet was received on or which the encapsulation was used, because the driver is responsible for setting `pak->network_start` to the start of the network layer data. Note that the data can be referenced on any alignment, although macros should be used to correct for any process alignment

dependencies. This code could actually be improved by reading the value of `vinesip->tc` into a local variable and performing the tests on the local copy instead of reading the value twice from shared memory.

```
vinesip = (vinesiptype *)pak->network_start;
dstnet = GETLONG(vinesip->ddstnet);
dsthost = GETSHORT(&vinesip->dsthost);
if (vinesip->tc & VINES_METRIC)
    return(FALSE);
if ((vinesip->tc & VINES_HOPS) == 0)
    return(FALSE);
```

Make the Forwarding Decision

Once the fast-switching routine has read the destination address and any other necessary data, it must determine whether the packet should be forwarded. This is done principally by looking up the destination address in a special cache, but it can also involve operations such as checking for the presence or absence of certain bits in a flags word.

If the packet is to be fast switched, the cache entry contains a pointer to the output interface and the encapsulation header to be used on that interface. The correct output routine is now called by one of the two methods described in the section “Software Architecture.”

If a cache entry is not found or any of the other tests fail, the packet cannot be fast switched and must be handed to the process level. The fast-switching routine returns `FALSE`, and the drivers ensure that the packet is sent to process level.

Transmit a Packet

The fast-switching output routine is responsible for modifying the received packet and transmitting it. It generally has a pointer to the packet and a pointer to the cache entry. The output routine must do the following:

1. Rewrite the packet encapsulation.
2. Reset the starting address and length of the packet.
3. Call the transmit routine for the output interface.

The following is an example of transmitting a packet; it is from the VINES code. Routines for the output of shared-memory fast-switching are usually this simple.

```
/*
 * First, set new starting location and length.
 */
pak->datagramstart = pak->network_start - E_ARPA_HDR_BYTES_OUT;
pak->datagramsize = E_ARPA_HDR_BYTES_OUT + pak->length;
if (pak->datagramsize < MINETHERSIZE)
    pak->datagramsize = MINETHERSIZE;

/*
 * Write the new header.
 */
macptr = (ulong *)pak->datagramstart;
cache_macptr = path->vinesp_mh.mac_longs;
PUTLONG(macptr++, *cache_macptr++);
PUTLONG(macptr++, *cache_macptr++);
PUTLONG(macptr++, *cache_macptr);
PUTSHORT((ushort *)macptr,TYPE_VINES);

/*
 * Now send the packet.
 */
(*path->idb->hwptr->fastsend) (path->idb->hwptr, pak);
return(TRUE);
```

21.2.2 Software Architecture

There are two types of software architecture for fast switching:

- Full Matrix
- Unique Routines

21.2.2.1 Full Matrix

In the full-matrix style of programming, there is one input routine for each specific encapsulation and each input routine knows how to rewrite the packet for each possible output routine. The input routine uses a `switch` statement to execute the correct output routine for a packet. This yields the best possible performance because the code can be fine-tuned for speed, but it grows in an order(n^2) fashion as new interfaces are added.

The following example of full-matrix programming is taken from the intracard IP fast-switching code. This `switch` clause and all the output routines are repeated for each possible input routine, with possible minor changes.

```
IP_FAST_STARTUP2
switch (output_interface) {
case FS_ETHER:
    /* Ethernet output code */
case FS_FDDI:
    /* FDDI output code */
case FS_TOKEN:
    /* Token Ring output code */
/* The rest of the encapsulation types follow. */
}
```

If you are implementing a new encapsulation on the router and it is the *n*th encapsulation routine, you need to write 2*n* new `case` statements. You also need to write one new input routine with a case for each possible output routine, and then in each of the existing input routines, you need to add a `case` statement for the new output encapsulation.

The following is another example from the IP intercard fast-switching code. Each of these sets of `if` clauses enumerates all possible encapsulations, with possible minor changes in the code executed. If you are implementing a new encapsulation on the router and it is the *n*th encapsulation routine, you will need to write 2*n* new `if` clauses. You need to write one new `if` clause with its embedded inner set of `if` clauses, and in each of the existing inner sets of `if` clauses you need to add a new `if` clause for the new output encapsulation.

```
if (output_interface & IDB_ETHER) {
    /* Output is Ethernet. */
    if (input_interface & IDB_ETHER) {
        /* Output Ethernet -- input Ethernet */
    } else if (input_interface & IDB_FDDI) {
        /* Output Ethernet -- input FDDI */
    } else ...
} else if (output_interface & IDB_FDDI) {
    /* Output is FDDI. */
    if (input_interface & IDB_ETHER) {
        /* Output FDDI -- input Ethernet */
    } else if (input_interface & IDB_FDDI) {
        /* Output FDDI -- input FDDI */
    } else ...
} else ...
```

21.2.2.2 Unique Routines

In this style of programming, there is one input routine for each specific encapsulation and one output routine for each specific encapsulation. The input routines call the correct output routines by indexing into a table of routines. This yields slightly lower performance although the code can be fine-tuned somewhat, but it is much easier to maintain as new interfaces are added. The code grows in an order(2*n*) fashion as new interfaces are added.

An example of this style is taken from the VINES fast-switching code:

```
return((*vfs_samemci[path->encaptype])(input, path));
```

If you are implementing a new encapsulation on the router and it is the *n*th encapsulation routine, you need to write two new routines. You need to write one new input routine that “removes” the incoming encapsulation and then vectors through the output routine table, and one new output routine that writes the new encapsulation. No other input or output routines need to be modified.

PART 5

Hardware-Specific Design

Porting Cisco IOS Software to a New Platform

One advantage of programs written in the C language is that they can be ported to a wide range of platforms. However, software in C can be written so that it is not portable to other platforms. This is true of parts of the Cisco IOS code, which assume a certain type of microprocessor. This chapter provides guidelines for writing Cisco IOS code that is portable to different types of CPUs.

When you write portable code, you need to be aware of the following issues:

- CPUs read data to and write data from memory using different methods. Some CPUs store data with the least-significant byte (LSB) first. This method is called little endian or low-high order. The Intel *x*86 family of CPUs use this method. Other CPUs store data with the most-significant byte (MSB) first. This method is called big endian or the high-low order. The Motorola 680*x*0 family of CPUs uses this method. Until now, released versions of the Cisco IOS software have run only on big-endian microprocessors (680*x*0 and MIPS). As a result, portions of the code assume a big-endian CPU.
- Some CPUs have strict rules about accessing memory on natural boundaries or even boundaries. Failing to follow the CPU's rules results in a fatal error.
- Integer sizes vary with different CPUs. Some integers are 16 bits, and others are 32 or 64 bits. Until now, the Cisco IOS software has been running on CPUs with 32-bit integers, and some portions of the code assume 32-bit integers.

This chapter discuss these and other issues in greater detail. There is also a section about various other portability issues that are less important but still need to be addressed when writing portable code. The remaining sections of the chapter present the methodology used at Cisco to handle the portability issues.

22.1 Portability Issues

This section discusses the following topics related to writing portable code:

- Byte Order
- Data Alignment
- C Pitfalls
- Other Portability Issues

22.1.1 Byte Order

Almost all portability problems are related to byte order. Portability demands that code does not depend upon the order of bytes in the host machine.There is a byte order defined as the *network byte order*. All data read in the network byte order must be canonicalized before it is used internally by the host machine. The data must also be re-ordered before it is sent to the network. The network byte order selected is the big-endian byte order.

Byte-ordering problems usually arise in the following areas:

- Unions
- Bit Fields
- Bit Operations
- Typecasting
- Character Constants

22.1.1.1 Unions

You should generally avoid using unions when writing portable code. To understand why this is the case, consider the `charlong` structure, which is defined in the Cisco IOS `types.h` file as follows:

```
typedef struct charlong_ {
    union {
        uc     harbyte[4];
        us    hortword[2];
        ul     onglword;
    } d;
} charlong;
```

Let's examine the behavior of this structure for big-endian and little-endian CPUs by assigning the byte stream `11 22 33 44` to the `cl` variable. On a big-endian CPU, this assignment would result in the following:

```
cl.d.byte[0] = 11; cl.d.byte[1] = 22; cl.d.byte[2] = 33; cl.d.byte[3] = 44;
```

The same assignment on a little-endian CPU would result in the following:

```
cl.dlbyte[0] = 11; cl.d.byte[1] = 22; cl.d.byte[2] = 33; cl.d.byte[3] = 44;
cl.d.sword = 2211; cl.d.sword[1] = 4433;
cl.d.lword = 44332211;
```

22.1.1.2 Bit Fields

Bit fields can cause problems for portability. For example, the `bpduhdrtype` structure is valid only for big-endian CPUs:

```
typedef struct bpduhdrtype_ {
    us                          hort protocol;/* protocol ID */
    uc                            har version;/* version identifier */
    uc                               har type;/* BPDU type */
    vola    tile uchar tc_acknowledgement: 1;/* topology change acknowledgment */
    volatile uchar notusedflags: 6;
    vo                    latile uchar tc: 1;/* topology change flag */
    uchar root_id[IDBYTES];
    uchar root_path_cost[4];
    uchar bridge_id[IDBYTES];
    uchar port_id[2];
    uchar message_age[2];
    uchar max_age[2];
    uchar hello_time[2];
    uchar forward_delay[2];
} bpduhdrtype;
```

In order for the `bpduhdrtype` structure to work the same on little-endian CPUs, it must be defined as follows. Note the difference in the bit field.

```
typedef struct bpduhdrtype_ {
    us                          hort protocol;/* protocol ID */
    uc                            har version;/* version identifier */
    uc                               har type;/* BPDU type */
    vo                    latile uchar tc: 1;/* topology change acknowledgment */
    volatile uchar notusedflags: 6;
    vola    tile uchar tc_acknowledgement: 1;/* topology change flag */
    uchar root_id[IDBYTES];
    uchar root_path_cost[4];
    uchar bridge_id[IDBYTES];
    uchar port_id[2];
    uchar message_age[2];
    uchar max_age[2];
    uchar hello_time[2];
    uchar forward_delay[2];
} bpduhdrtype;
```

The problem becomes more difficult when the bit field spans byte boundaries. In this case, you can use bit masks to access a given field. For example, the `bpduhdrtype` structure could be written as follows:

```
typedef struct bpduhdrtype_ {
    us                          hort protocol;/* protocol ID */
    uc                            har version;/* version identifier */
    uc                               har type;/* BPDU type */
    vola          tile uchar topology_change;/* topology change acknowledgment */
    uchar root_id[IDBYTES];
    uchar root_path_cost[4];
    uchar bridge_id[IDBYTES];
    uchar port_id[2];
    uchar message_age[2];
    uchar max_age[2];
    uchar hello_time[2];
    uchar forward_delay[2];
} bpduhdrtype;
```

The following `#define` statements could be defined:

```
#de fine TC_ACKNOWLEDGEMENT0x01
#d                 efine TC0x80
```

The following code, which is nonportable, tests and sets the acknowledgment bit:

```
if (bpdu->tc_acknowledgement)
    topology_change_acknowledged(span);

bpdu->tc_acknowledgement = port->topology_change_acknowledge;
```

This code could be rewritten with bit masks as follows:

```
if ((bpdu-topology_change & TC_ACKNOWLEDGEMENT) == TC_ACKNOWLEDGEMENT)
    topology_change_acknowledged(span);

bpdu->topology_change = (bpdu->topology_change & ~TC_ACKNOWLEDGEMENT) |
                        (port->topology_change & TC_ACKNOWLEDGEMENT);
```

22.1.1.3 Bit Operations

Pay special attention to code that performs bit operations. Often, bitwise operations can be made more portable by using the bitwise and (&), bitwise or (|) and bitwise complement (~) operators as seen in the example in the "Bit Fields" section.

22.1.1.4 Typecasting

When writing portable code, you should question each typecasting occurrence. Code such as the following gives different results on CPUs with different byte orders when you dereference `p`:

```
char *p;
short n;

p = (char *)&n;
```

22.1.1.5 Character Constants

Because of byte order differences, multicharacter constants are handled differently. The following is an example of portable code for defining characters to be used as a string:

```
char crlf[] = "\r\n";
```

22.1.2 Data Alignment

Some CPUs require strict data alignment, and if an item is not aligned, references to the data cause a trap that aborts the program. In the same fashion, assembler instructions must also be aligned to the instruction length or a trap occurs. The Motorola 680*x*0 CPUs require strict data alignment. Normally, the compiler takes care of data alignment, but referencing data in a byte stream might cause a trap if the data is misaligned.

22.1.3 Data Siz

Portable code should use predefined types for data that will be exchanged with other machines. The sizes of some data types, in particular `int`, differ from one CPU to the next. For this reason, there should be no code dependencies on the `int` size of the CPU.Take extra care on machines where the `int` and `long` are the same size. Failure to do so results in code that is difficult to port to smaller machines.

There are three appropriate usages for portable code:

- Return values. Functions can return values of type `int`. In fact, many C standard library functions return an `int`.
- Function parameters.
- Register integer variables.

In all these case, the Cisco IOS code might assume that an `int` is at least 16 bits long.

22.1.4 C Pitfalls

Certain aspects of the C language depend on the implementation, which is not necessarily the same on all platforms. This section covers those aspects.

22.1.4.1 EnumTypes

Do not assume that `enum` symbols are necessarily contiguous.

22.1.5 Other Portability Issues

The following issues also have an impact on writing portable code:

- Performance
- Stack Usage and Stack Growth
- Compliance with Encapsulations

22.1.5.1 Performance

Code that is based on the architecture of the CPU or the operating system creates problems when ported to CPUs or operating systems with a different architecture. For example, the Cisco IOS code is a nonpreemptive operating system, and most of the code assumes this, even though it should not. If you port part of the code to a preemptive operating system, you will encounter resource protection problems.

Consider the following portion of code, which accesses a queue to add a new element. On Cisco's current platforms, this code functions correctly. However, if this code were to run on a platform such as Windows NT, the operating system could suspend the queuing operation to give CPU time to another task. This task might also access the same data area, which would result in problems with the first queuing operation.

```
add code example here
```

22.1.5.2 Stack Usage and Stack Growth

Not all architectures have the same stack usage scheme. For example:

```
main()
{
    long parm1, parm2, parm3, parm4, parm5
    do_something(parm1, parm2, parm3, parm4, parm5);
}

do_something(long arg)
{
    long *ptr;
    long var1, var2, var3, var4, var5;

ptr = &arg;

var1 = ptr[0];
var2 = ptr[1];
var3 = ptr[2];
var4 = ptr[3];
var5 = ptr[4];
}
```

This code has the following problems with regard to portability:

- The code assumes that the stack grows down. Not all architectures have the stack grow from the high address to the low address, while some have it grow in the opposite direction, from the lower memory address to the higher.
- The code assumes that arguments are pushed from right to left on the stack. Some architectures pass the argument from left to right.
- The code assumes that all argument are pushed on the stack. In some architectures, the first few arguments are passed in registers to speed the call.

The best approach to managing stack usage is to use the macros in the Cisco IOS `stdarg.h` header file to implement a variable parameter function call. In an ANSI C environment, these macros in `stdarg.h` are in `varargs.h`.

It is also important to pay close attention to the return value of functions. Some systems return the results of a function call in a different register depending on the type of the function result. For example, the m68k ELF format uses register d0 to return ordinal values (`cha` , `unsigned cha` , `short`, `unsigned short`, `long`, and `unsigned long`), but register a0 might be used for returning addresses (`char *`, `void *`, `short *`, and so on).

22.1.5.3 Compliance with Encapsulations

To allow code to be easily ported, you must follow the spirit of the protocol header definition. Do not cheat and assume you are running on a big-endian CPU.

The following code assumes that the processor is big-endian and does not care about longword alignment:

```
*((long *)pak->datagramstart) = HDLC_BRIDGECODE;
```

This code would be better written as the following portable code:

```
PUTSHORT(hdlc->var_hdlcflags,HDLC_STATION);
PUTSHORT(hdlc->var_hdlctype,TYPE_BRIDGE);
```

Now let's look at some code that handles alignment issues but totally disregards endian issues. The header definition is as follows:

```
/*
 * 802.10 SDE encapsulation header data structure.
 */
typedef struct sdehdrtype_ {
    uchar sde_dsap;
    uchar sde_lsap;
    uchar sde_control;
    uc                      har sde_said[4];/* Security association ID */
    uc                       har sde_sid[8];/* Station ID */
    uc                        har sde_flags;/* Flags */
    uc                       har sde_fid[4];/* Fragment ID */
} sdehdrtype;
```

The following code writes the header to a buffer:

```
/*
 * Write an 802.10 SDE header to an network buffer.
 *
 */
static inline
void tbridge_bfr_sde_header_wr_inline (idbtype *outputsw, uchar *bfr_wr_ptr,
                                       uchar *src_addr)
{
    ulong said;
    if (!outputsw->sde_said_db)
        return;
    said = outputsw->sde_said_db->sdb_said;
    /*
     * Write SDE header.
     */
    PUTSHORT(bfr_wr_ptr, (SAP_SDE | SAP_SDE << 8));
    bfr_wr_ptr += 2;
    PUTSHORT(bfr_wr_ptr, (LLC1_UI << 8 | said >> 24));
    bfr_wr_ptr += 2;
    PUTSHORT(bfr_wr_ptr, (ushort)(said >> 8));
    bfr_wr_ptr += 2;
    PUTSHORT(bfr_wr_ptr, (((uchar)(said & 0xFF)) << 8 | *src_addr));
    bfr_wr_ptr += 2;
    PUTLONG(bfr_wr_ptr, GETLONG(&src_addr[1]));
    bfr_wr_ptr += 4;
    /*
     * Flag is zero for now - no fragmentation support
     */
    PUTLONG(bfr_wr_ptr, (*((uchar *)(&src_addr[5]))) << 24);
}
```

22.2 Cisco's Implementation of Portability

This section discusses some features of the Cisco IOS code that enable the writing of portable code.

22.2.1 Inline Assembler

Some assembler code is needed on every platform, commonly to accelerate the processing or to perform some function that the C language does not allow. Pay attention to where these assembler routines are stored in the development tree.You must store inline assembler routines in processor-dependent files.

22.2.2 Header Files

The development tree contains header files for each platform and processor currently supported by Cisco IOS software. These header files are in the `sys/machine` directory. They use the following naming conventions:

- `cpu_`*`processor`*`.h`—Platform-independent definitions for the specified processor
- `cisco_`*`platform`*`.h`—Platform-dependent definitions for the specified processor
- `cisco_`*`processor`*`.h`—Cisco-specific definitions for the specified processor

22.2.3 Byte-Order Functions

The Cisco IOS software contains many macros for reordering bytes, including the following, which are useful for reordering bytes in a packet:

```
ORDER_BYTE_SHORT(addr)
ORDER_BYTE_LONG(addr)
```

These two macros reorder the 16-bit word and the 32-bit word at the address specified in `addr`, if necessary. The reordered bytes are stored at the same address.

22.2.4 Endian #defines

The following `#define` statements are used to define the byte order:

```
#define BIGENDIAN 1234
#define LITTLEENDIAN 4321
```

These two `#define` statements are defined for every platform. To find out which byte order the current platform uses, the code must refer to `BYTE_ORDER`. This `#define` is assigned the proper endian `#define` based on the platform. Using this approach, source code can refer to `BYTE_ORDER` without using the preprocessor.

Byte-ordering code could look something like this:

```
if (BYTE_ORDER == BIGENDIAN){
    do_something();
else
    do_something_else();
```

The optimizer removes the unneeded code from the executable format so that no running time is wasted determining the byte order and the code remains easy to read without the `#ifdef` where endian-dependent code must be used.

Another `#define` statement used to define byte order is `ORDER_DATA_CMD`. This macro can be used to called the canonicalize routines (see the “Canonical Functions” section in this chapter). For example, the following macro executes the parameter only if byte reordering is necessary:

```
ORDER_DATA_CMD(ip_canonicalize(pak));
```

When defining data structures, compare `BYTE_ORDER` in an `#if` statement. For example, in the following code, no other endian values should be used. All other `#define` statements have been removed from the code, including those for `LITTLE_ENDIAN` and `BIG_ENDIAN`.

```
#if BYTE_ORDER == BIGENDIAN
#define SOME_MACRO SOME_BIG_ENDIAN_DEFINE
#endif

#if BYTE_ORDER == LITTLEENDIAN
#define SOME_MACRO SOME_LITTLE_ENDIAN_DEFINE
#endif
```

Although bit fields still exist in the code mainly because of the amount of code that would need to change, you should avoid using them. Masking is a portable approach.

22.2.5 GET and PUT Macros

Each platform has a set of `GET` and `PUT` macros to extract and insert words and double words in byte streams.

The following are the `GET` macros. They take as input the address where the `short` or `long` value should read, and they return the requested `short` or `long` value.

```
GETSHORT(address)
GETLONG(address)
```

The following are the `PUT` macros. They take as input the address where the `short` or `long` value should be placed and the value that should be placed at that address.

```
PUTSHORT(address, value)
PUTLONG(address, value)
```

22.2.6 Canonical Functions

The approach taken to port Cisco IOS software to little-endian platforms is to canonicalize the packets when they enter and exit from each layer. The canonical functions are invoked only when needed, based on endians using the `BYTE_ORDER #define`. You should define these functions as `static inline`.

The following is an example of a canonical function from the IP code:

```
static inline void ip_canonicalize (iphdrtype *ip)
{
ORDER_BYTE_SHORT(&ip->tl);
    ORDER_BYTE_SHORT(&ip->id);
    ORDER_BYTE_LONG(&ip->srcadr);
    ORDER_BYTE_LONG(&ip->dstadr);
    /*
     * Check for any IP options.
     */
    if (ltob(ip->ihl) >= MINIPHEADERBYTES) {
        int i, len;
        uchar *opts = (uchar *)&ip[1];
        for (i = 0; i < ltob(ip->ihl) - MINIPHEADERBYTES; i += len) {
            len = ip_option_len(&opts[i], ltob(ip->ihl) - MINIPHEADERBYTES);
            if (len == 0)
            return;
            switch (opts[i]) {
            case IPOPT_RRT:
            case IPOPT_LSR:
            case IPOPT_SSR:
            {
            ipopt_routetype *ptr = (ipopt_routetype *)&opts[i];
                int j, nhops = btol(ptr->length - IPOPT_ROUTEHEADERSIZE);
                if ((nhops < 1) || (nhops > btol(MAXIPOPTIONBYTES)))
                    return;
                for (j = 0; j < nhops; j++)
                    ORDER_BYTE_LONG(&ptr->hops[j]);
                break;
            }
            case IPOPT_TSTMP:
            {
                ipopt_tstmptype *ptr = (ipopt_tstmptype *)&opts[i];
                int j, ntimes = btol(ptr->length - IPOPT_TSTMPHEADERSIZE);
                if ((ntimes < 0) || (ntimes > btol(MAXIPHEADERBYTES)))
                    return;
                for (j = 0; j < ntimes; j++)
                    ORDER_BYTE_LONG(&ptr->tsdata[j]);
                break;
            }
            case IPOPT_SID:
            {
                ipopt_sidtype *ptr = (ipopt_sidtype *)&opts[i];
                if (ptr->length != IPOPT_SIDSIZE)
                    return;
                ORDER_BYTE_SHORT(ptr->streamid);
                break;
            }
        }
    }
    }
}
```

PART 6

Management Services

CHAPTER 23

Command-Line Parser

23.1 Overview: Parser

The Cisco IOS command-line parser is a finite state machine described by a series of macros that define the sequence of a command's tokens. Each macro defines a node in the state diagram of a command. This definition includes a pointer to the node to process if the current node matches the command-line input and an alternate node to process regardless of whether the current node is accepted. Optional parameters and keywords are indicated by alternate states in the parse tree. A macro exists for every type of object that can be parsed, such as keywords, integers, addresses, and string text.

23.1.1 Traversing the Parse Tree

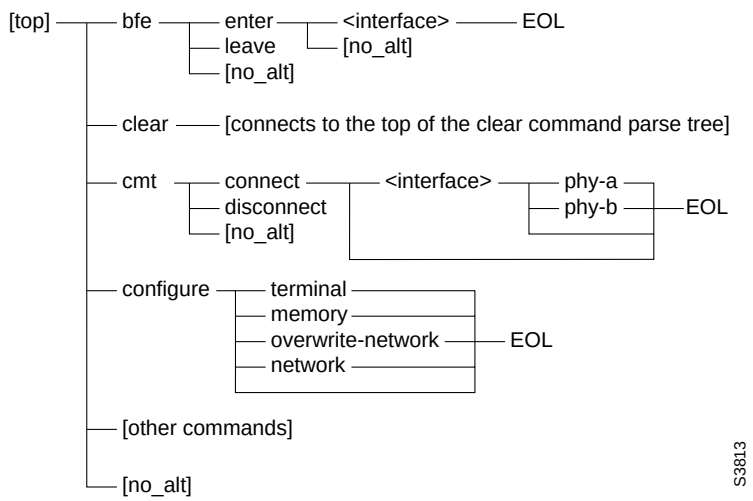
When parsing a command, the command-line parser checks the entire parse tree, searching all alternates for a given token, in order to detect ambiguous input that might occur when abbreviated keywords are used. Although this might seem to present a significant load to the CPU, very few branches of the parse tree are actually traversed for any given command input.

When generating nonvolatile output, which builds an expression that describes how a platform is configured, the entire parse tree must be traversed to accurately reflect the current state of the platform. This is because other mechanisms, such as SNMP, can be used to modify the router's internal state. Even when traversing the entire parse tree, it takes only about 1 second to generate a normal configuration and about 5 to 8 seconds to generate large configurations on a slower router.

Figur e23-1 shows a global view of the parse tree for a subset of the Cisco IOS router EXEC commands. In this figure, the accepting nodes are distributed to the right, and alternate nodes are distributed down the figure on the left. When traversing the parse tree, the parser does the following:

1. The parser begins at [top] and compares the first input token with all the alternates listed on the left, starting with *bfe*, *clear*, and so on.
2. When a match occurs, the parser transitions to the accepting node on the right.
3. The parser allocates a new console status block (CSB).
4. The parser searches the remaining alternate nodes to identify ambiguous commands.
5. After the entire parse tree has been searched, if there has only been one command match, the saved CSB is restored and the specified command function is called with the CSB as its argument. The command function extracts the parsed values from the CSB.

Figure 23-1 Traversing the Parse Tree



For example, if the input is **configure network**, the parser traverses the parse tree as follows:

1. Each of the alternates starting at [top] is checked.
2. When the parser reaches **configure**, which matches, the parser advances the input pointer and checks the tokens accepted from **configure**, which are listed to the right of **configure**.
3. The keyword **network** is matched, followed by a match of end of line (EOL).
4. When EOL is reached, the state of the parse is stored on a stack. This state includes parsed values and a pointer to a command function that is to be called to execute the command.

23.1.2 Transition Structure

Each node in a command's state diagram is defined by a transition structure, which is created by the parser macros. The transition structure has the following format:

```
typedef const struct transition transition;
struct transition_ {
    transition *accept;
    transition *alternate;
    const trans_func function;
    const void * const arguments;
};
```

function is a function pointer that parses a token. It takes two arguments:

- A pointer to the console status block (CSB), which is a `parseinfo` structure that contains the parser state
- A pointer to the transition

function pushes the *alternate* transition onto the parser stack, and if the token is parsed correctly, *function* pushes the *accept* transition onto the stack.

arguments is a pointer to a token-specific structure that contains information about how the token should be parsed, where parsed information should be stored, and help strings.

23.2 Build Parse Trees

You build parse trees for a command using a set of macros to describe each token in the command. Each macro defines a unique node in a command's state diagram.

23.2.1 Construction of Parse Trees

You typically build the parse tree for each command in a separate file. Because C does not allow forward referencing without explicit declaration of the forward referenced item, you define parse trees in bottom-up order.

23.2.1.1 Example: Construction of Parse Trees

The following example of constructing a parse tree shows the parse tree for the **disable** EXEC command:

```
/*
 * disable
 */

EO    LS(exec_disable_endline, enable_command, exec_disable_endline);
KEYWORD (exec_disable, exec_disable_endline, ALTERNATE,
         "disable", "Turn off privileged commands", PRIV_ROOT);
```

In this example, the `KEYWORD` macro takes the following arguments:

- `exec_disable` is the transition structure that is the entry point to the parse tree
- `exec_disable_endline` is the name of the accept transition. This is the node to which control is passed if the word "disable" is matched in the input.
- `ALTERNATE` is the name of the alternate state.
- `"disable"` is the keyword to match.
- `"Turn off privileged commands"` is the long help string to provide to the user
- `PRIV_ROOT` is a flag word that determines how the keyword is handled. Table 2 3 -2 lists the possible privilege levels.

If the user input begins with the word **disable**, the `KEYWORD` macro processes the keyword and takes the accept transition to the `exec_disable_endline` node, which checks for end of line. If end of line is found, the parser saves the function pointer `enable_command`, along with any other state of the parse. The third argument to `EOLS` is stored in `csb->which` before calling the named function, `exec_disable_endline`. This argument allows a family of commands to share a single processing function.

Note the coding style of the arguments in the `KEYWORD` macro. `KEYWORD` is the first item on a line so that it is easy to find when you scan parse tree code.

23.2.2 Parse a Keyword Tok

To parse a keyword token, use the `GENERAL_KEYWORD` and `KEYWORD_*` macros. Tabl e23-1Tab le23-1 lists some common macros for parsing keywords.

Table 23-1 Macros for Parsing Keywords

Parsing Task	Macro
Parse a keyword.	`GENERAL_KEYWORD(`*`name, accept, alternate, keyword, help, privilege, variable, value, match, flags`*`)`
Parse a keyword, accepting partial matches and ignoring case.	`KEYWORD(`*`name, accept, alternate, keyword, help, privilege`*`)`
Parse a keyword, matching a minimum number of characters.	`KEYWORD_MM(`*`name, accept, alternate, keyword, help, privilege, count`*`)`
Parse a keyword without parsing the trailing white space.	`KEYWORD_NOWS(`*`name, accept, alternate, keyword, help, privilege`*`)`
Parse a keyword and optionally parse the trailing white space.	`KEYWORD_OPTWS(`*`name, accept, alternate, keyword, help, privilege`*`)`
Parse a keyword that is followed by an existing interface.	`INTERFACE_KEYWORD(`*`name, accept, alternate, variable, valid_ifs, keyword, help`*`)`

The following parameters are common to most of the macros listed inTa ble23-1:

- *`name`* is the name of this node in the parse tree. This name is used to link nodes together.
- *`accept`* specifies the accept transition. It is the name of the node to process next if the keyword token matches.
- *`alternate`* is the name of the alternate node to process. The alternate node transition is always taken, regardless of whether this node is accepted. It is this linkage that allows all alternates to be checked at a given point in the parsing process.
- *`keyword`* is the keyword token itself.
- *`help`* specifies a help string that explains the keyword token or the set of options that depend on the token's presence.
- *`privilege`* is a flag word that encodes the privilege level required to execute the command and other options, such as whether help is provided and visible to users, whether NV generation is performed, and other parser control functions.

  Table 23-2Table 2 3 -2 lists the flags for specifying the keyword privilege level. You can modify them with the **privilege** configuration command. Specify only one privilege level for a keyword.

  Table 23-3Table 2 3 -3 lists the flags for specifying how the keyword should be parsed and NV generated, and how it should provide help. Several values can be `OR`ed together for the *`privilege`* argument.

Table 23-2 Flags for Specifying Privilege Level When Parsing Keywords

Flag	Description
PRIV_MIN	Set the privilege level needed to parse the keyword to the lowest privilege level (0). This is useful for keywords that should always be available to a user, regardless of their privilege level, such as keywords that disable, enable, end, exit, and provide help.
PRIV_USER	Set the privilege level needed to parse the keyword to 1. This is the default user privilege level.
PRIV_ROOT	Set the privilege level needed to parse the keyword to the maximum privilege level (15). This is the default privilege level for the **enable** command.
PRIV_MAX	Same as PRIV_ROO .

Table 23-3 Flags for Specifying Other Options When Parsing Keywords

Flag	Description
PRIV_INTERNAL	The command is an internal command and is unavailable unless you enter the **service internal** configuration command.
PRIV_UNSUPPORTED	The command is unsupported and hidden, and no help is provided. If this command is entered, a warning message is displayed indicating that the command is unsupported; the command is then executed.
PRIV_USER_HIDDEN	The keyword is hidden to users with a privilege level of PRIV_USER or less unless the user has entered the **terminal full-help** EXEC command or **full-help** line global configuration command.
PRIV_SUBIF	The keyword is available on subinterfaces. If the keyword does not have this flag, it is unavailable when configuring subinterfaces.
PRIV_HIDDEN	The keyword is hidden, and all keywords following the accept transition are also hidden. No help for any of these keywords is displayed, and the parse tree following the keyword is not searched. This keyword prevents the generation of help output for entire parse trees. An example of a hidden command is the **write core** command.
PRIV_DUPLICATE	The keyword occurs twice in the current mode, and this instance is hidden. No help is provided for the duplicate instance, but the parse tree under it is still searched. For example, the keyword **xns** appears twice as a global configuration command, once for the protocol configuration and again for global routing configuration. The second instance is flagged as PRIV_DUPLICATE.
PRIV_NONVGEN	The keyword is skipped during NV generation. This is useful for obsolete commands that must still be accepted, but that have been replaced with newer commands. Using this flag in combination with either PRIV_HIDDEN or PRIV_NOHELP allows obsolete keywords to be gracefully deprecated. To aid in the transition to the new command, you need to modify the help message of the obsolete command to warn of the change.
PRIV_NOHELP	The keyword is hidden. This flag affects the help output of the given keyword only; all successive keywords are processed normally, including the generation of help messages. There are currently no examples of this in the Cisco IOS parser.

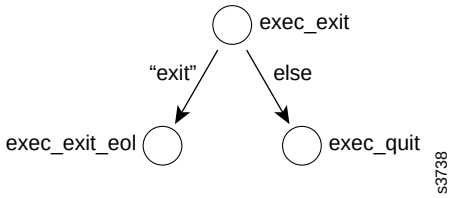
23.2.2.1 Example: Parse a KeywordToken

The following example parses the **exit** keyword token. In this example, the current node is named exec_exit, the accept transition points at the exec_exit_eol node, the alternate transition points at the exec_quit node, the token itself is exit, the help string is "Exit from the EXEC," and the privilege level is set to the minimum level.

```
KEYWORD(exec_exit, exec_exit_eol, exec_quit, "exit", "Exit from the EXEC",
        PRIV_MIN);
```

Figur e23-2 illustrates the transition diagram for this example, showing the three nodes exec_exit, exec_exit_eol, and exec_quit.

Figure 23-2 Transition Diagram for Parsing a Keyword



This example of using the `KEYWORD` macro to parse the **exit** keyword creates the following transition structure. In this structure, the parser calls `keyword_action`, which pushes the alternate transition, `PARSER_exec_quit`, onto the stack. If the keyword and trailing white space are parsed correctly, the parser then pushes the accept transition, `PARSER_exec_exit_eol`, onto the stack. `Lexec_exit` is a `keyword_struct` structure that contains the keyword, help, and privilege-level information.

```
transition PARSER_exec_exit = {
    &PARSER_exec_exit_eol,
   &PARSER_exec_quit,
    keyword_action,
    &Lexec_exit
};
```

23.2.3 Parse a Number Token

To parse a number token, use the `GENERAL_NUMBER`, `NUMBER`, and other macros.Tabl e23-4Tab le23-4 lists some common macros for parsing number tokens.

Table 23-4 Macros for Parsing Numbers

Parsing Task	Macro
Parse a number in a specified range.	`GENERAL_NUMBER(`*`name`*`, `*`accept`*`, `*`alternate`*`, `*`variable`*`, `*`lower`*`, `*`upper`*`, `*`help`*`, `*`flags`*`)`
Parse a decimal, an octal, or a hexadecimal number in a specified range.	`NUMBER(`*`name`*`, `*`accept`*`, `*`alternate`*`, `*`variable`*`, `*`lower`*`, `*`upper`*`, `*`help`*`)`
Parse a decimal, an octal, or a hexadecimal number in a specified range without parsing trailing white space and without providing help.	`INUMBER(`*`name`*`, `*`accept`*`, `*`alternate`*`, `*`variable`*`, `*`lower`*`, `*`upper`*`)`
Parse a decimal number in a specified range.	`DECIMAL(`*`name`*`, `*`accept`*`, `*`alternate`*`, `*`variable`*`, `*`lower`*`, `*`upper`*`, `*`help`*`)`
Parse a decimal number in a specified range without parsing trailing white space and without providing help.	`IDECIMAL(`*`name`*`, `*`accept`*`, `*`alternate`*`, `*`variable`*`, `*`lower`*`, `*`upper`*`)`
Parse an octal number in the range 0 through 0xFFFFFFFF.	`OCTAL(`*`name`*`, `*`accept`*`, `*`alternate`*`, `*`variable`*`, `*`help`*`)`
Parse an octal number in the range 0 through 0xFFFFFFFF without parsing trailing white space and without providing help.	`IOCTAL(`*`name`*`, `*`accept`*`, `*`alternate`*`, `*`variable`*`)`
Parse a hexadecimal number in the range 0 through 0xFFFFFFFF.	`HEXNUM(`*`name`*`, `*`accept`*`, `*`alternate`*`, `*`variable`*`, `*`lower`*`, `*`upper`*`, `*`help`*`)`
Parse a hexadecimal number in a specified range.	`HEXDIGIT(`*`name`*`, `*`accept`*`, `*`alternate`*`, `*`variable`*`, `*`lower`*`, `*`upper`*`, `*`help`*`)`

Parsing Task	Macro
Parse a hexadecimal number in the range 0 through 0xFFFFFFFF without parsing trailing white space and without providing help.	HEXADECIMAL(*name*, *accept*, *alternate*, *variable*)

The following parameters are common to most of the macros listed in Table 23-4:

- *name* is the name of this node in the parse tree. This name is used to link nodes together.
- *accept* specifies the accept transition. It is the name of the node to process next if the keyword token matches.
- *alternate* is the name of the alternate node to process. The alternate node transition is always taken, regardless of whether this node is accepted. This linkage allows all alternates to be checked at a given point in the parsing process.
- variable is the variable in the CSB in which to store the number
- lower is the lower bound of the number range.
- upper is the upper bound of the number range.
- *help* specifies a help string that explains the keyword token or the set of options that depend on the token's presence.

23.2.3.1 Example: Parse a Number Token

The following example parses a number in the range 1 through 15. In this example, the current node is named enable_level, the accept transition points at the enable_endline node, the alternate transition points at the enable_endline node, 1 and 15 specify the number range, and the help string is "Enable level."

```
NUMBER(enable_level, enable_endline, enable_endline, OBJ(int,1), 1, 15,
       "Enable level");
```

This example of using the NUMBER macro creates the following transition structure. In this structure, the parser calls general_number_action, which pushes the alternate transition, PARSER_enable_endline, onto the stack. If the number and trailing white space are parsed correctly, the parser pushes the accept transition, PARSER_enable_endline, onto the stack. Lenable_level is a number_struct structure that contains an offset into the CSB to store the number parsed, the range within which the number must lie, and the help string.

```
transition PARSER_enable_level = {
    &PARSER_enable_endline,
    &PARSER_enable_endline,
    general_number_action,
    &Lenable_level
};
```

23.2.4 Parse a Keyword-Number Combination

A keyword followed by an integer value is a common occurrence in the Cisco IOS commands. To parse this combination, you can use the KEYWORD, NVGENS, NOPREFIX, NUMBER, and EOLS macros. Because keyword-number combinations are so common, two macros are provided explicitly for parsing them: PARAMS and PARAMS_KEYONLY. These macros combine the functions of the KEYWORD, NVGENS, NOPREFIX, NUMBER, and EOLS macros.

```
PARAMS(name, alternate, keyword, variable, lower, upper, function, subfunction,
       keywordhelp, variablehelp, privilege)

PARAMS_KEYONLY(name, alternate, keyword, variable, lower, upper, function,
               subfunction, keywordhelp, variablehelp, privilege)
```

23.2.4.1 Examples: Parse a Keyword-Number Combination

The following example shows how to use the PARAMS macro instead of the KEYWORD, NVGENS, NUMBER, and EOLS macros:

```
PARAMS(snark_cmd, ALTERNATE, "snark",
       OBJ(int,1), 0, 10,
       snark_command, SNARK,
       "Snark command", "Snark number", PRIV_ROOT);
```

If you use the KEYWORD, NVGENS, NUMBER, and EOLS macros, you need to define the parse tree for this command as follows:

```
EOLS(snark_eols, snark_command, SNARK);
NUMBER(snark_number, snark_eols, no_alt, OBJ(int,1), 0, 10, "Snark number");
NVGENS(snark_nvgen, snark_number, snark_command, SNARK);
KEYWORD(snark_kw, snark_nvgen, ALTERNATE, "snark", "Snark command", PRIV_ROOT);
```

The following example shows how to use the PARAMS_KEYONLY macro instead of the KEYWORD, NVGENS, NOPREFIX, NUMBER, and EOLS macros:

```
PARAMS_KEYONLY(snark_cmd, ALTERNATE, "snark",
               OBJ(int,1), 0, 10,
               snark_command, SNARK,
               "Snark command", "Snark number", PRIV_ROOT);
```

If you use the KEYWORD, NVGENS, NOPREFIX, NUMBER, and EOLS macros, you need to define the parse tree for this command as follows:

```
EOLS(snark_eols, snark_command, SNARK);
NUMBER(snark_number, snark_eols, no_alt, OBJ(int,1), 0, 10, "Snark number");
NOPREFIX(snark_no, snark_number, snark_eols);
NVGENS(snark_nvgen, snark_number, snark_command, SNARK);
KEYWORD(snark_kw, snark_nvgen, ALTERNATE, "snark", "Snark command", PRIV_ROOT);
```

23.2.5 Parse Optional Keywords

A common command syntax allows for several possible keywords at a given point in the parse. An example is the **arp** interface configuration command, which has the following syntax:

[**no**] **arp** { **arpa** | **probe** | **smds** | **snap** | **timeout** *seconds* | **ultranet** }

The parser code for this command is as follows. Reading from the bottom up, each of the **arp** command keywords is checked against the input. If one matches, the transition to the associated EOLS causes the current state of the parse to be saved. For example, if the input is **arp s**, the "smds" and "snap" keywords both match and two parse states are saved. The parser reports the input as ambiguous. The "timeout" keyword demonstrates the use of the PARAMS_KEYONLY macro.

```
/*
 * arp { arpa | probe | smds | snap | timeout <seconds> | ultranet }
 * no arp { arpa | probe | smds | snap | timeout [<seconds>] | ultranet }
 *
 * csb->which = ARP_ARPA, ARP_PROBE, ARP_SNAP or ARP_ENTRY_TIME
 * OBJ(int,1) = seconds for ARP_ENTRY_TIME
 *
 */

/* arp ultranet */
EOLS    (ci_arp_ultra_eol, arpif_command, ARP_ULTRA);
KEYWORD (ci_arp_ultra, ci_arp_ultra_eol, no_alt, "ultranet", "",
PRIV_CONF|PRIV_HIDDEN);

/* arp timeout <seconds> */
PARAMS_KEYONLY (ci_arp_timeout, ci_arp_ultra, "timeout", OBJ(int,1), 0, -1,
                arpif_command, ARP_ENTRY_TIME, "Set ARP cache timeout", "Seconds",
                PRIV_CONF);

/* arp snap */
EOLS    (ci_arp_snap_eol, arpif_command, ARP_SNAP);
KEYWORD (ci_arp_snap, ci_arp_snap_eol, ci_arp_timeout,
         "snap", "IEEE 802.3 style arp", PRIV_CONF);

/* arp smds */
EOLS    (ci_arp_smds_eol, arpif_command, ARP_SMDS);
KEYWORD (ci_arp_smds, ci_arp_smds_eol, ci_arp_snap,
         "smds", "", PRIV_CONF|PRIV_HIDDEN);

/* arp probe */
EOLS    (ci_arp_probe_eol, arpif_command, ARP_PROBE);
KEYWORD (ci_arp_probe, ci_arp_probe_eol, ci_arp_smds,
         "probe", "HP style arp protocol", PRIV_CONF);

/* arp arpa */
EOLS    (ci_arp_arpa_eol, arpif_command, ARP_ARPA);
KEYWORD (ci_arp_arpa, ci_arp_arpa_eol, ci_arp_probe,
         "arpa", "Standard arp protocol", PRIV_CONF);

KEYWORD (ci_arp, ci_arp_arpa, ALTERNATE, "arp",
         "Set arp type (arpa, probe, snap) or timeout", PRIV_CONF);
```

23.2.6 Parse Mixed String and Nonstring Tokens

When a string token can be accepted at the same place that a nonstring token can be matched, a specific parse order must be followed. The rules for this situation are as follows:

- The nonstring token must be checked first, before checking the string token.
- A TEST_MULTIPLE_FUNCS macro must precede the STRING macro. The TEST_MULTIPLE_FUNCS macro tests whether any of the nonstring variables have matched. If they have, do not attempt to parse the token as a string, because the parse would always match, resulting in an ambiguous command.

23.2.6.1 Example: Parse Mixed String and Nonstring Tokens

The following example uses the **ping** command to illustrate how to parse mixed string and nonstring tokens. The **ping** command has the following syntax:

ping [[*hint*] *destination*]

The parser code for this command follows. In this code, first all protocol keywords are checked. To determine whether any protocol keyword matches, the test is performed. Finally, the destination, which is taken as a string variable that is later handled by any protocol-specific processing, is checked.

```
/*
 * ping [[hint] <destination>]
 *
 * OBJ(string,1) = destination
 * OBJ(int,1) = protocol hint (if any)
 */

EO    LS(exec_ping_eol, ping_command, 0);

ST  RING(exec_ping_destination, exec_ping_eol, exec_ping_eol,
         OBJ(string,1), "Ping destination address or hostname");

TEST_MULTIPLE_FUNCS(exec_ping_test, exec_ping_destination, no_alt, NONE);

KEYWORD_ID (exec_ping_vines, exec_ping_destination, exec_ping_test,
            OBJ(int,1), PING_HINT_VINES, "vines", "Vines echo", PRIV_USER);
KEYWORD_ID (exec_ping_apollo, exec_ping_destination, exec_ping_vines,
            OBJ(int,1), PING_HINT_APOLLO, "apollo", "Apollo echo", PRIV_USER);
KEYWORD_ID (exec_ping_novell, exec_ping_destination, exec_ping_apollo,
            OBJ(int,1), PING_HINT_NOVELL, "novell", "Novell echo", PRIV_USER);
KEYWORD_ID (exec_ping_atalk, exec_ping_destination, exec_ping_novell,
            OBJ(int,1), PING_HINT_ATALK, "atalk", "Appletalk echo", PRIV_USER);
KEYWORD_ID (exec_ping_clns, exec_ping_destination, exec_ping_atalk,
            OBJ(int,1), PING_HINT_CLNS, "clns", "CLNS echo", PRIV_USER);
KEYWORD_ID (exec_ping_xns, exec_ping_destination, exec_ping_clns,
            OBJ(int,1), PING_HINT_XNS, "xns", "XNS echo", PRIV_USER);
KEYWORD_ID (exec_ping_pup, exec_ping_destination, exec_ping_xns,
            OBJ(int,1), PING_HINT_PUP, "pup", "PUP echo", PRIV_USER);
KEYWORD_ID (exec_ping_ip, exec_ping_destination, exec_ping_pup,
            OBJ(int,1), PING_HINT_IP, "ip", "IP echo", PRIV_USER);

KEYWORD (exec_ping, exec_ping_ip, ALTERNATE, "ping", "Send echo messages", PRIV_USER);
```

23.2.7 Process "No" Commands

A command can have a "no" prefix as long as it has been added to the Config command tree. Your code to add a command to the Config command tree would be something like this:

```
>static parser_extension_request xyz_chain_init_table[] = {
>    { PARSE_ADD_CFG_TOP_CMD, &pname(xyz_cfg_commands) },
>    { PARSE_ADD_EXEC_CMD, &pname(xyz_exec_commands) },
>    { PARSE_ADD_SHOW_CMD, &pname(xyz_show_commands) },
>    { PARSE_LIST_END, NULL }
>
>};
>
>static void xyz_parser_init (void)
>{
>    parser_add_command_list(xyz_chain_init_table, "XYZ support");
>}
```

Note You must explicitly add a command to the Config tree for the “no” prefix to be recognized. In the example above, if the **show** command had not been added to the tree, a **no show** command would have been truly no- show.

This means that you can enter a no-prefixed command only in Config mode. This poses no problem because, with one exception, only the Config commands need the “no” prefix. The exception is **debug**, the only command outside Config mode that can h ve a “no” prefix. The parse tree defined for **debug** (`/vob/ios/sys/parser/exec_debug.h`) explicitly defines the “no” keyword:

```
EOLNS   (exec_debug_help_eol, debug_help_command);
PRIV_TEST(exec_debug_help, exec_debug_help_eol, NONE,
          exec_debug_commands, PRIV_MIN | PRIV_HIDDEN);

KEYWORD_ID(exec_debug_false, exec_debug_help, no_alt,
           sense, FALSE,
           "debug", "Disable debugging functions (see also ’undebug’)",
           PRIV_OPR);

KEYWORD (exec_debug_no, exec_debug_false, ALTERNATE,
         "no", "Disable debugging functions", PRIV_OPR);

KEYWORD_ID(exec_debug_true, exec_debug_help, exec_debug_no,
           sense, TRUE,
           "debug", "Debugging functions (see also ’undebug’)",
PRIV_OPR);

#undef  ALTERNATE
#define ALTERNATE       exec_debug_true
```

23.2.7.1 csb->sense

The parser provides a uniform method for handling the “no” version of commands. When the “no” keyword has been parsed, `csb->sense` is set to FALSE; otherwise, it is set to TRUE.

You insert the `NOPREFIX` macro into the parse chains to identify where in the parse any successive tokens can be safely ignored when the “no” version of a command has been entered. This allows the user to prefix any command with the word **no**. The `NOPREFIX` macro skips to the end of the input line before transitioning to a designated node in the parse chain. Because the parser allows any command to be prefixed with **no**, all command functions should test the state of `csb->sense` and act accordingly.

23.2.7.2 Example: Process “No” Commands

The following example uses the **appletalk zip query** command to illustrate how to use the `NOPREFIX` macro. This command has the following syntax:

appletalk zip-query-interval *interval*
no zip-query-interval [*interval*]

When the user specifies the "no" version of this command, anything entered after the **zip-query-interval** keyword is ignored.

```
/*
 * appletalk zip-query-interval <interval>
 * no appletalk zip-query-interval [<interval>]
 *
 * OBJ(int,1) = interval
 */
EO     LS(cr_at_zonequery_eol, appletalk_command, ATALK_ZONEQUERY);
NU   MBER(cr_at_zonequery_val, cr_at_zonequery_eol, no_alt,
          OBJ(int,1), 1, -1, "Seconds");
NOPREFIX (cr_at_nozonequery, cr_at_zonequery_val, cr_at_zonequery_eol);
NV   GENS(cr_at_zonequery_nvgen, cr_at_nozonequery,
          appletalk_command, ATALK_ZONEQUERY);
KE  YWORD(cr_at_zonequery, cr_at_zonequery_nvgen, cr_at_arp,
          "zip-query-interval", "Interval between ZIP queries", PRIV_CONF);
```

23.2.8 Nonvolatile Output Generation

When generating nonvolatile output, which is an expression that describes how a platform is configured, the entire parse tree must be traversed to accurately reflect the current state of the platform.

Nonvolatile (NV) generation is performed by traversing the parse tree and calling the command function to output the data associated with a given command. As the parse tree is traversed, the keywords of each command are stored in the text string `csb->nv_command`. This string is used by the command function when generating the NV output. Storing keywords in `csb->nv_command` eliminates many of the character strings used for NV generation in the command functions.

There is one limitation to this mechanism. Any command that must retrieve data from within the router must call the command function before the macro that parses the data item. For example, for the command **ip route** *address1 address2*, the `ip_route_command` function must be called prior to *address1* when doing NV generation. This function takes the string "ip route" from `csb->nv_command` and traverses (walks) the routing table, generating the routing table entries. A special macro, `NVGENS`, tests whether NV generation is being performed. It calls `action_func` if this is true; otherwise, it transitions to alternate.

23.3 Link Parse Trees

The `#define` of `ALTERNATE` allows parse trees to be stored in separate files, yet be linked together when included via `#include` statements in the `chain.c` file.

23.3.1 Example: Link Parse Trees

The following example shows the linkage between parse trees defined in the two files `exec_disable.h` and `exec_disconnect.h`. Remember that the parse trees are read from bottom to top.

In the file `exec_disconnect.h`, `ALTERNATE` is redefined to be `exec_disconn`, which is the name of the **disconnect** `KEYWORD` macro. In `exec_disable.h`, `ALTERNATE` is used as the name of the alternate node to process after checking for the **disable** keyword. When these two files are included via `#include` in the `chain.c` file in the order shown, `ALTERNATE` in `exec_disable.h` becomes `exec_disconn`, thus establishing the linkage between the two commands.

exec_disconnect.h

```
/*
 * disconnect [ <connection-number> | <connection-name> ]
 *
 * OBJ(int,1) = <connection-number>
 * OBJ(string,1) = <connection-name>
 */
EO    LS(exec_disconn_eol, disconnect_command, 0);

ST  RING(exec_disconn_name, exec_disconn_eol, NONE,
         OBJ(string,1), “The name of an active telnet connection”);
TESTVAR (exec_disconn_name_test, exec_disconn_name, NONE,
         NONE, NONE, exec_disconn_eol, OBJ(int,1), 0);
NU  MBER(exec_disconn_num, exec_disconn_eol, exec_disconn_name_test,
         OBJ(int,1), 1, MAX_CONNECTIONS,
         “The number of an active telnet connection”);
KEYWORD (exec_disconn, exec_disconn_num, ALTERNATE,
         “disconnect”, “Disconnect an existing telnet session”, PRIV_USER);

#u  ndefALTERNATE
#d     efine ALTERNATEexec_disconn
```

exec_disable.h

```
/*
 * disable
 */
EO    LS(exec_disable_endline, enable_command, CMD_DISABLE);
KEYWORD (exec_disable, exec_disable_endline, ALTERNATE,
         “disable”, “Turn off privileged commands”, PRIV_ROOT);

#u  ndefALTERNATE
#d     efine ALTERNATEexec_disable
```

23.4 Manipulate CSB Object

23.4.1 Overview: CSB Objects

The CSB stores parsed data in a set of objects, which provides a generic way to reference the parser variables. If a variable changes, only the macro needs to be changed instead of changing each reference to the variable.You specify parser variables with the *variable* argument in the parser macros.

There are two naming mechanisms for CSB objects:

- OBJ—Used in the parse chains to set an object value.
- GETOBJ—Used in the command functions to access the object value.

Both naming mechanisms specify the data type and instance identifier (number) of the stored object. Table 23-5 lists the allowable object data types and numbers.

Table 23-5 Data Type and Number of Stored CSB Objects

Data type	Type name	Storage type	Number of objects
Unsigned integers	int	uint	1-22
String	string	char *	1-6

Data type	Type name	Storage type	Number of objects
IDB (interface descriptor block)	`idb`	`idbtype *`	1
PDB (protocol descriptor block)	`pdb`	`void *`	1
CDB (controller descriptor block)	`cdb`	`cdbtype *`	1
Protocol address	`paddr`	`addrtype *`	1-10
Hardware address	`hwaddr`	`hwaddrtype *`	1-4

23.4.2 Examples of CSB Objects

The following examples demonstrate the use of `OBJ` and `GETOBJ` in the parser.

In the following example, specifying `OBJ(int,1)` in a `NUMBER` macro stores the parsed integer in the first unsigned integer object:

```
NUMBER(snark_number, snark_eol, no_alt, OBJ(int,1), 1, 10, “Snark from 1 to 10”)
```

In the following example, specifying `GETOBJ(int,1)` in a `NUMBER` macro returns the unsigned integer:

```
NUMBER(snark_number, snark_eol, no_alt, GETOBJ(int,1), 1, 10, “Snark from 1 to 10”)
```

In the following example, specifying `OBJ(paddr,10)` in the `IPADDR` macro stores the parsed IP address in the tenth protocol address object:

```
IPADDR(boojum_ipaddr, boojum_ipaddr_eol, no_alt, OBJ(paddr,10), “An IP address”)
```

In the following example, specifying `GETOBJ(paddr,10)` in the `IPADDR` macro returns an `addrtype` pointer to the parsed address:

```
IPADDR(boojum_ipaddr, boojum_ipaddr_eol, no_alt, GETOBJ(paddr,10), “An IP address”)
```

23.5 Add Commands Dynamically

When Cisco IOS subset images are created or when the user configures the router, the command-line parser should display only the commands that make sense in that subset image or configuration. You specify this using link points, which are locations in the parse tree. Link points allow the partial loading of commands—the commands are added at run time rather than when the Cisco IOS code is compiled.

23.5.1 Create a Link Point

To create a link point to which to add commands at run time, use the `LINK_TRANS` macro:

```
LINK_TRANS(name, accept)
```

23.5.1.1 Example: Create a Link Point

The following example creates the link point identified by the transition `option_extend_here` to which additional commands can be added at run time:

```
LINK_TRANS(option_extend_here, no_alt);
KEYWORD(option2, option2_accept, option_extend_here,
        “2option”, “Second option”, PRIV_ROOT);
KEYWORD(option1, option1_accept, option2,
        “1option”, “First option”, PRIV_ROOT);
```

To uniquely identify the link point, you must also add an `enum` to `h/parser.h`:

```
enum {
  PARSE_LIST_END=0,
  PARSE_ADD_EXEC_CMD,
  ...
  PARSE_ADD_OPTION_CMD
};
```

23.5.2 Register a Link Point with the Parser

After you create a link point, use the `parser_add_link_point()` function to register it with the parser:

```
boolean parser_add_link_point(int which_chain, const char *module,
                              transition *lp);
```

23.5.2.1 Example: Register a Link Point with the Parser

The following example registers a link point with the parser:

```
parser_add_link_point(PARSE_ADD_OPTION_CMD, “option command”,
                      &pname(option_extend_here);
```

23.5.3 Display Registered Link Points

To display link points that have been registered with the parser, use the **show parser links** hidden EXEC command:

show parser links [*link-name*]

The following is sample output from the **show parser links** command:

```
Router# show parser links
Current parser link points:
  Na                              me    ID        AddrType
  en                          d         of                  list00x01
  ex                              e    c1     0x5CF041
  .
  .
  .
  op                    ti    on           commandXXX0xXXXXX1
```

23.5.4 Link Commands to a Link Point

To link parser commands to a link point at run time, use the `parser_add_command_list()` function:

```
boolean parser_add_command_list(const parser_extension_request * chain,
                                const char *module);
```

23.5.4.1 Example: Link Commands to a Link Point

The following example adds a command from the `snark` subsystem to the `option` link point:

```
#define ALTERNATE NONE
#include “option_snark.h”
LINK_POINT(snark_option_commands, ALTERNATE);
#undef ALTERNATE

static const parser_extension_request snark_chain_init_table[] = {
  { PARSE_ADD_OPTION_CMD, &pname(snark_option_commands) },
  { PARSE_LIST_END, NULL }
};

void snark_parser_init (void)
{
  parser_add_command_list( snark_chain_init_table, “snark”);
}
```

To display the subsystems that have added commands to a link point, use the **show parser links** *link-name* EXEC command.

```
Router# show parser link points option command
Current links for link point 'option command':
  snark
```

23.5.5 Create Link Exit Points

Link exit points are similar to link points, except that the parser goes from many commands to one link point instead of going from one link point to many added commands. You use the `parser_add_link_exit()` function instead of the `parser_add_link_point()` function. Link exit points are useful for adding options in the middle of a command and then returning.

The `parser_add_link_exit()` function has the following format:

```
boolean parser_add_link_exit(int which_chain, const char *module,
                             transition *lp);
```

23.5.5.1 Example: Create Link Exit Points

The following example uses the code for the **snmp enable traps** global configuration command to illustrate link exit points.

First, the link point and exit are created and registered with the parser:

```
LINK_TRANS(conf_snmp_enable_return_here, conf_snmp_enable_trap_opts);
LINK_TRANS(conf_snmp_enable_extend_here, NONE);
...
void snmp_parser_init (void)
{
  parser_add_link_point(PARSE_ADD_CFG_SNMP_ENABLE_CMD,
                        "config snmp trap/inform",
                        &pname(conf_snmp_enable_extend_here));
  parser_add_link_exit(PARSE_ADD_CFG_SNMP_ENABLE_EXIT,
                       "config snmp trap/inform exit",
                       &pname(conf_snmp_enable_return_here));
}
```

Other subsystems can then add options to the command:

```
LINK_EXIT(cfg_snmp_enable_isdn_exit, no_alt);
KEYWORD_OR(cfg_snmp_enable_isdn, cfg_snmp_enable_isdn_exit, NONE,
           OBJ(int,1), (1<<TRAP_ENABLE_ISDN),
           "ISDN", "Enable SNMP ISDN traps", PRIV_ROOT);
LINK_POINT(cfg_snmp_enable_isdn_entry, cfg_snmp_enable_isdn);

const static parser_extension_request isdn_chain_init_table[] = {
    { PARSE_ADD_CFG_SNMP_ENABLE_CMD, &pname(cfg_snmp_enable_isdn_entry) },
    { PARSE_ADD_CFG_SNMP_ENABLE_EXIT,
      (dynamic_transition *) &pname(cfg_snmp_enable_isdn_exit) },
    { PARSE_LIST_END, NULL }
};

void isdn_parser_init (void)
{
    parser_add_command_list(isdn_chain_init_table, "ISDN");
}
```

23.6 Manipulate Parser Modes

23.6.1 Add a Parser Mode

You can add new parser modes using the `parser_add_mode()` function:

```
parser_mode *parser_add_mode (const char *name, const char *prompt,
                              const char *help, boolean do_aliases,
                              boolean do_privileges, const char *alt_mode,
                              mode_save_var_func save_vars,
                              mode_reset_var_func reset_vars,
                              transition *top, transition *nv_top)
```

When a command is parsed to enter a new mode, the `set_mode_byname()` function is used to change the parser to that mode.

If a parse fails in the new mode, the parser calls the `save_vars()` function to save mode state information and then tries to parse the command using *`alt_mode`* mode. If the command is parsed in *`alt_mode`* mode, the parser changes the current mode to *`alt_mode`* mode, unless the command parsed changes the mode. If the command is not parsed in *`alt_mode`* mode, the parser calls the `reset_vars()` function to reset mode state information and remain in the current mode.

23.6.1.1 Example: Add a Parser Mode

The following example adds a new configuration mode named `boojum`. The new mode has the prompt `Router(config-boojum)#`. The mode can have aliases, and the privilege level of commands in the mode can be changed. The first transition in this mode is `boojum_top`.

```
parser_mode *mode;

mode = parser_add_mode("boojum", "config-boojum", "Boojum configuration mode",
                       TRUE, TRUE, "configure", NULL, NULL,
                       &pname(boojum_top), &pname(boojum_top));
```

23.6.2 Add an Alias to a Mode

To add a default alias to a mode, use the `add_default_alias()` function:

```
void add_default_alias(parser_mode *alias, const char *name,
                       cons t c har*command);
```

23.6.2.1 Example: Add an Alias to a Mode

The following example adds an alias b which is expanded to boojum in snark mode.

```
add_default_alias(snark_mode, "b", "boojum");
```

CHAPTER 24

Writing, Testing, and Publishing MIBs

The Simple Network Management Protocol (SNMP) is the language for communication between a managing system running a network management application and a managed system running an agent. Between them they share the concept of a Management Information Base (MIB) that defines the information that the agent can make available to the manager.

MIBs and an agent are commonly provided in networked systems to allow remote observation and control using management applications on other systems.

This chapter provides an overview of SNMP and MIBs and describes a general procedure for establishing a new MIB, attempting in the process to answer questions commonly asked by MIB developers and to address mistakes commonly made by developers. Writing MIBs is more of an art than an exact programming science, so this chapter cannot begin to provide a design and programming solution for every possible case. The last section of this chapter describes the procedure for testing and then publishing a MIB.

Most of the information in this chapter was gathered from the Cisco SNMP Web page. As a rule, the Web page has more information about development specifics and is more up-to-date:

http://wwwin-eng.cisco.com/Eng/IOS/SNMP_WWW/cisco-snmp.html

If you have questions about designing, writing, and testing MIBs that are not answered by this chapter or the SNMP Web site, contact the e-mail alias mib-consulting.

24.1 SNMP Overview

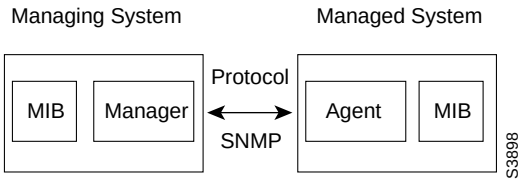
The section presents a high-level overview of SNMP. The discussion in this section is applicable to both SNMPv1 and SNMPv2. This section discusses the following topics:

- Internet Network Management Framework: Definition
- MIB: Definition
- SMI: Definition
- Transport Protocols
- SNMP Facilities
- Asynchronous Notifications

24.1.1 Internet Network Management Framework: Definition

SNMP network management is based on the Internet Network Management Framework. This framework defines a model in which a managing system called a *manager* communicates with a managed system. The manager runs a network management application, and the managed system runs an *agent*, which answers requests from the manager. The manager converses with the agent using the Simple Network Management Protocol (SNMP). Figure24- 1 illustrates the Internet Network Management Framework model.

Figure 24-1 Internet Network Management Framework Model



24.1.2 MIB: Definition

The Management Information Base (MIB) defines all the information about a managed system that a manager can view or modify. The MIB is located on the managed system and can consist of standard and proprietary portions.

The agent and manager each have their own view of the MIB. The agent presents the contents of the MIB and knows how to retrieve that information. The manager might use a MIB description to kno what to expect in a given MIB and might store that information in a translation that it prefers.

For more information about MIBs, see the section “MIB Concepts” later in this chapter.

24.1.3 ASN.1: Definition

Abstract Syntax Notation 1 (ASN.1) is the formal language used by SNMP. ASN.1 consists of two parts, one part referred to as ASN.1 and a second part called Basic Encoding Rules (BER).

You use ASN.1 to describe SNMP MIBs. Specifically, you use the subset of ASN.1 that is defined in the SNMP Structure of Management Information (SMI). ASN.1 is a human-readable language that can also be understood by machines through a MIB compiler.

BER is a method for taking information that is defined with ASN.1 and encoding it in a system-independent way so that it can be transmitted between computers (typically, between managers and agents) across a network. BER has specific rules for how to encode integers, text strings, and other values. BER is a machine-readable language.

ASN.1 and BER are defined in the International Telecommunication Union (ITU) Recommendations ASN.1, X.208, and BER X.209.

24.1.4 SMI: Definition

The SNMP Structure of Management Information (SMI) defines the components of a MIB and the formal language for describing them. The components include the following:

- Sections of a MIB description, which include
  - Setup

  - Data objects
  - Notifications
  - Conformance requirements
- Data types for the information in the MIB, which are
  - Basic computer forms, such as integer and octet string
  - SNMP-specific forms, such as a counter or gauge

For more information about the SMI, see the section "SMI Overview" in this chapter.

24.1.5 Transport Protocols

SNMP can be carried over a wide variety of transport protocols. The most common combination is UDP over IP. Other possibilities include AppleTalk, NetWare, and raw Ethernet.

24.1.6 SNMP Facilities

SNMP has security facilities that identify the requester and the operational context in which a request can be performed by the agent. Examples of operational context are read-only or read-write access, providing a MIB subset for a particular group of users, and obtaining a MIB subset from another location or through another mechanism such as by proxy.

SNMP MIB management operations allow SNMP to observe and control MIB information. These operations consists of reading (using `get` operations) and modifying (using a `set` operation). These operations allow you to get read-only information and get or set read-write information depending on your identity and the context you can reach.

24.1.7 Asynchronous Notifications

In most SNMP interactions, a manager makes a request to which an agent responds. It is also possible for agents to proactively provide information to a manager and for managers to provide information to each other. This is done using asynchronous notifications.

SNMP has two types of asynchronous notifications:

- Traps—Unacknowledged datagrams that are sent by the agent to the manager
- Informs—Acknowledged datagrams that are sent from one manager process to another

24.2 MIB Concepts

24.2.1 MIB: Overview

Most SNMP development and use centers around the Management Information Base (MIB). An SNMP MIB is an abstract data base, that is, it is a conceptual specification for information that a management application can read and modify. The SNMP agent translates between the internal data structures and formats of the managed system and the external data structures and formats defined for the MIB.

The SNMP MIB is organized as a tree structure with conceptual tables. Relative to this tree structure, the term MIB is used in two senses. In one sense, a MIB is a branch of the MIB tree. A branch usually contains information about a single aspect of technology, such as a transmission medium or a routing protocol. In this sense, a MIB is more accurately called a *MIB module*, which is usually defined in a single document. In the second sense, the term MIB refers to a collection of MIB modules. Such a collection might comprise, for example, all the MIB modules implemented by a given agent or the entire collection of MIB modules defined forSNMP.

24.2.2 Standard and Enterprise MIBs

MIBs can be standard or enterprise (proprietary). Internet-standard MIBs are defined by working groups of the Internet Engineering Task Force (IETF) and published as Requests for Comment (RFCs). Enterprise MIBs are defined by other organizations, usually individual companies. They instrument technology not covered by standard MIBs, either completely or as an extension to a standard MIB.

24.2.3 MIB-I and MIB-II

There are several revisions of the SNMP MIB standard. The original revision, which is referred to as MIB-I, is obsolete. It was followed by a second revision, referred to as MIB-II.

MIB-II contains branches for the basic areas of instrumentation, such as the system, its network interfaces, IP, and TCP. The initial specification of the MIB-II standard defined all these areas in a single MIB module. However, as SNMP evolves, portions of this MIB are being updated in technology-specific MIB modules, for example the TCP-MIB and UDP-MIB modules.

24.2.4 Agent Implementations

An agent implementation is defined by well-defined compliance groups in MIB modules and the agent capabilities specified in RFC 2580. Neither the MIB description nor the agent capabilities definition can be used alone to predict the abilities of an agent.

MIB modules define compliance groups in their ASN.1. Compliance groups are collections of objects from a MIB module that make up a logical subset of the MIB that might be mandatory, conditional, or optional in compliant implementations. An agent capabilities definition specifies which compliance groups an agent implements.

Compliance groups, in combination with `AGENT-CAPABILITIES` specified in RFC 2580, define the implementation of an agent, including variations in individual MIB objects. A MIB description cannot be used to predict the abilities of an agent. An agent capabilities definition comes closer, but ultimately an application must be able to smoothly deal with whatever it receives in response to its requests, for example, because a part of the MIB might be disabled through management control.

24.2.5 MIB Objects

24.2.5.1 Object: Definition

A MIB *object*, also sometimes called a *variable*, is a leaf in the MIB tree. Each leaf represents an individual item of data. Examples of objects are counters and protocol status. Leaf objects are connected to branch points.

The SNMP framework uses object somewhat differently than Open System Interconnection (OSI) management. An OSI object is a network entity, such as a router or a protocol, that has attributes. These OSI attributes and SNMP objects are essentially the same concept, that is, they both represent individual data values.

24.2.5.2 Lexicographic Ordering of Objects

The objects in the MIB tree are sorted using lexicographic ordering. This means that object identifiers are sorted in sequential, numerical order. Lexicographic ordering is important when using the GetNext protocol operation, because this operation takes an object identifier (OID) or a partial OID as input and returns the next object from the MIB tree base on the lexicographic ordering of the tree.

24.2.5.3 Object Identifier: Definition

An object is uniquely identified by the list of branch points that extends from the top of the MIB tree down to the leaf, composing an *object identifie* . The final part of the OID, the *instance identifie* , designates the specific occurrence of an object. This means that an object identifier designates each leaf object and branch point in the tree. An object can have one or more instance identifiers. The instance identifier for an ordinary object that has a single instance (that is, a scalar object) is always 0. Objects that compose conceptual tables have instance identifiers with other values to identify the row in the table.

An object identifier is expressed as a series of integers or text strings. Technically, the numeric form is the *object nam* and the text form is the *object descriptor.* In practice, both are called *object identifiers*, or *OIDs*. The numeric form is used in the protocols among machines. The text form, sometimes mixed with the numeric form, is for use by people.

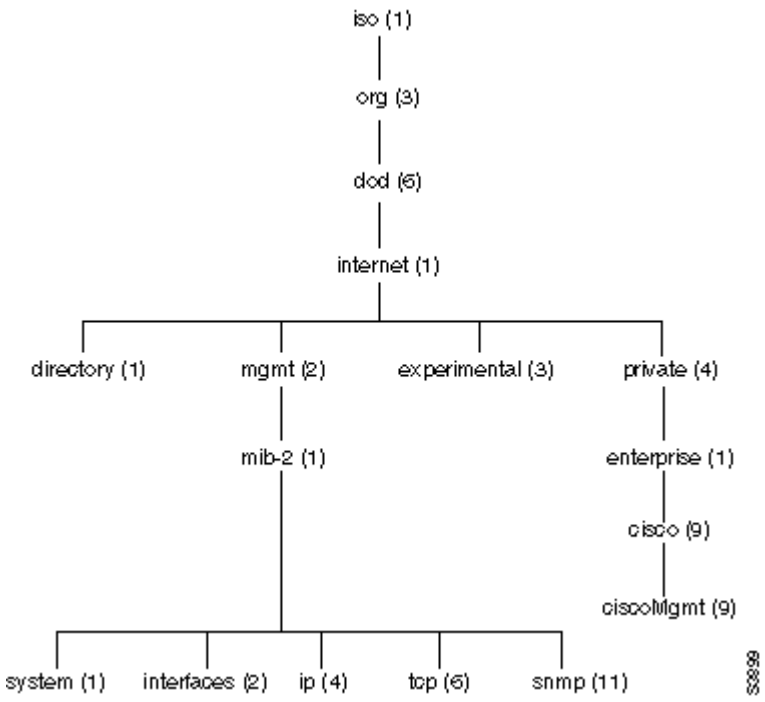
Figur e24-2 illustrates the lexicographic ordering of MIB object identifiers. The root of the MIB tree is iso, which has a numeric identifier of 1. The following OIDs refer to the system branch point in the MIB tree and are logically identical:

```
iso.org.dod.internet.mgmt.mib-2.system
1.3.6.1.2.1.1
iso.org.dod.internet.2.1.1
```

One of the objects in the `system` branch is `sysDesc` . Its full OID can be one of the following:

```
iso.org.dod.internet.mgmt.mib-2.system.sysDescr.0
1.3.6.1.2.1.1.1.0
```

Figure 24-2 MIB Object Identifiers



24.2.6 SNMP Conceptual Tables

24.2.6.1 SNMP Conceptual Tables: Definition

SNMP conceptual tables are the mechanism for defining a set of objects that appear repeatedly, indexed by some entry name**.** Tables can contain simple objects only; they cannot contain other tables.

SNMP conceptual tables have a rigid structure, as defined in the SMI.

An entry, or row, in a table specifies a set of objects for the same instance. The row is uniquely identified by one or more *table indexes*, or *auxiliary objects*. The OID of an object that is stored in a table consists of the OID for that object's position in the MIB tree concatenated with a representation of all the table indexes for an entry in the table. The table indexes thus compose the instance identifier.

Each row is the set of objects for a particular instance, such as its state, speed, and description. Each column is the objects of the same type for all instances, such as all the speeds.

24.2.6.2 Simple SNMP Conceptual Tables

An example of a simple SNMP table is the `ifTable`, which is a key table in the interface MIB and is defined in RFC 1573. This table is simple because it has a single, integer index object, `ifIndex`. The index object of the `ifTable` key table is `ifIndex`, which is defined as an integer. The OID for a counter from the `ifTable` can be one of the following, which are semantically identical:

```
iso.org.dod.internet.mgmt.mib-2.interfaces.ifTable.ifEntry.ifInOctets
1.3.6.1.2.1.2.2.1.10
```

Adding the instance identifier for `ifIndex 7` gives the following OID:

```
iso.org.dod.internet.mgmt.mib-2.interfaces.ifTable.ifEntry.ifInOctets.7
1.3.6.1.2.1.2.2.1.10.7
```

In OIDs, the row selection, which represents the instance of an object, follows the column selection. This OID structure can be confusing when you apply the principle of lexicographic ordering to a table. Note, however, that if you use the GetNext protocol operation to walk a table, the operation proceeds column by column rather than row by row. That is, the operation returns all the instances in one column before starting on the next column.

24.2.6.3 Complex SNMP Conceptual Tables

Complex tables are those with multiple indexes or variable-length indexes, or a combination of the two. The following example from the Cisco VINES MIB uses multiple indexes, including one that is of variable length. The `INDEX` clause from the ASN.1 definition is the following. In this example, The first two indexes are simple integers, with `ifIndex` being imported from the standard `ifTable`, and the third index is a variable-length octet string.

```
INDEX { cvForwNeighborHost, ifIndex, cvForwNeighborPhysAddress }
```

24.2.6.4 Coding Index Objects

Coding the integers in the index is simple and obvious.

Coding the variable-length index object is more complex. RFC 2578 contains rules for encoding variable-length index objects as instances. The general rule is that the value is preceded by a length, and the length and each part of the value are separate subidentifiers. For example, if you have a neighbor host number of 9, `ifIndex 3`, and an Ethernet neighbor physical address of 0000.0c03.1ef0, the following is the instance portion of an object for that row:

```
9.3.6.0.0.12.3.30.240
```

In RFC 2578, SNMPv2 extends the instance encoding rules to include an IMPLIED keyword, which can be used on the final instance of a variable-length object. When this keyword is present, that part of the instance does not have a length in front of it.

Lexicographic ordering for variable-length instance objects with a count effectively sorts the object by length. This means that an ASCII text index with a length is not in alphabetical order.

Index objects are often defined as part of the table, in the first positions, but this is not necessary. They can be defined in another table, or in another MIB module, as long as they are referenced appropriately in the table's index clause.

In SNMPv2, index objects are not accessible because retrieving them is redundant: The OID for any object in a table by definition includes all the index objects. An application can therefore extract the appropriate index object values as a by-product of retrieving another object. Having index objects be inaccessible avoids problems with the meaning of read-write index objects and makes the `GetBulk` operation more efficient by not retrieving large numbers of unnecessary objects.

24.2.6.5 Tables Insideof Tables

SNMP does not allow you to nest tables inside of tables. However, you can achieve this effect with multiple table indexes. Using this method, the table that would have been inside another table has the indexes of the first table as its own first indexes.

For example, the standard repeater MIB, defined in RFC 1516, organizes ports into groups. This MIB defines the index clause for the group table as follows, which allows each group in the group table to contain a port table:

```
IN    DEX{ rptrGroupIndex }
```

To make the port table a subtable of the group table, its index clause is defined as follows. In this example, the object `rptrPortGroupIndex` is defined to be equivalent to the values of `rptrGroupIndex`.

```
IN    DEX{ rptrPortGroupIndex, rptrPortIndex }
```

It is not necessary to redefine the object in this way. Instead, you can reuse the existing object by defining the index clause as follows. This method is preferable.

```
IN    DEX{ rptrGroupIndex, rptrPortIndex }
```

24.3 SMI Overview

The SNMP Structure of Management Information (SMI) defines the data structures and operations of MIBs and the formal language for describing them. Part of the technique for doing this selects and uses a subset of ASN.1.

RFC 1155 defines the first version of the SMI, commonly referred to as SMIv1. RFC 1902, and later RFC 2578, updated the SMI. This second version is commonly referred to as SMIv2. This newer version of the SMI includes some new data types and some significant modifications to the macros used to describe MIB modules.

24.3.1 Primitive Data and Application Types

The SMI recognizes primitive data and application types. In most cases, you must use these data types as they are defined. The only aggregated type or ASN.1 `SEQUENCE` a MIB can contain is a conceptual table constructed according to rules discussed in the section "SNMP Conceptual Tables" earlier in this chapter.

The SMI recognizes the following ASN.1 primitive data types:

- `INTEGER`—Signed integer in the range -2,147,483,648 to 2,147,483,647
- `OCTET STRING`—String of bytes of length 0 to 65,535
- `OBJECT IDENTIFIER`—Numeric ASN-1-type object identifier

The SMI recognizes the following ASN.1 application types for general use:

- `IpAddress`—Fixed-length, 4-byte Internet address
- `Counter32`—Unsigned, wrapping 32-bit integer in the range 0 to 4,294,967,295
- `Gauge32`—Unsigned, nonwrapping 32-bit integer in the range 0 to 4,294,967,295
- `Unsigned32`—Unsigned, nonwrapping 32-bit integer in the range 0 to 4,294,967,295; this is indistinguishable from `Gauge32`.
- `TimeTicks`—Unsigned 32-bit integer in the range 0 to 4,294,967,295 representing hundredths of seconds since an epoch started
- `Counter64`—Unsigned, wrapping 64-bit integer in the range 0 to 18,446,744,073,709,551,615

You can hide the primitive data types under textual conventions, as defined in RFC 2579 and described in the section "Textual Conventions" later in this chapter.

24.3.2 Textual Conventions

SNMP and the SMI do not allow you to add data types. However, you can use the textual conventions defined in RFC 2579 to hide the data types. It is sometimes preferable to hide data types in this way, because the textual conventions can carry formatting hints not available to the basic data types. Also, the textual conventions make MIBs clearer, promote common solutions to common problems, and provide additional object-handling information to applications.

Textual conventions allow you to supply a different name for and additional information about a primitive data or application type. For example, the standard `DisplayString` is based on `OCTET STRING`. `DisplayString` is limited to a maximum of 255 bytes rather than the 65,535 bytes allowed by `OCTET STRING` and prints only Network Virtual Terminal (NVT) ASCII characters as defined in RFC 854.

Textual conventions can lead to common, human-readable definitions for objects with the same semantics. This means that applications can implement general handling for a textual convention and know which objects to apply it to. An example of this is the standard `RowStatus`, which defines an entire state machine for adding and deleting rows in tables. Textual conventions accomplish these useful ends without adding to actual encoding of the protocol. Whatever can be done with a textual convention, the information transmitted in the protocol is in the form of the underlying, real data type.

The following are the standard textual conventions as defined in RFC 2579:

- `DisplayString`—Represents textual information taken from the NVT ASCII character set. Objects defined using `DisplayString` cannot be longer than 255 characters.
- `PhysAddress`—Represents media-level or physical-level addresses.
- `MacAddress`—Represents an 802 MAC address in the canonical order as defined by IEEE 802.1a. That is, the address is formatted as if it were transmitted least-significant bit first.
- `TruthValue`—Represents a boolean value.
- `TestAndIncr`—Represents integer information used for atomic operations. When the management protocol specifies that an object instance having this syntax is to be modified, the new value supplied by the management protocol must precisely match the value currently held by the instance. If not, the management protocol `Set` operation fails. If the current value is the maximum value of 2^31 - 1 (2,147,483,647 decimal), the value held by the instance is wrapped to 0; otherwise, the value held by the instance is incremented by 1.
- `AutonomousType`—Represents an independently extensible type identification value. For example, `AutonomousType` can be used to indicate a particular subtree that contains further MIB definitions, or it can be used to define a particular type of protocol or hardware.
- `VariablePointer`—Is a pointer to a specific object instance, for example, `sysContact.0` or `ifInOctets.3`.
- `RowPointer`—Is a pointer to a conceptual row. The value is the name of the instance of the first accessible columnar object in the conceptual row. For example, `ifIndex.3` would point to the third row in the `ifTable`. Note that if `ifIndex` were not accessible, `ifDescr.3` would be used instead.
- `StorageType`—Is an indication of how a row is stored in memory. It includes the types `volatile`, `nonVolatile`, `permanent`, and `readOnl` .
- `TDomain`—Is a type of transport service, such as UDP or TCP.

- `TAddress`—Is a transport service address, such as an IP address.
- `RowStatus`—Manages the creation and deletion of conceptual rows.
- `TimeStamp`—Is the value of the MIB-II `sysUpTime` object at which a specific occurrence happened. The specific occurrence must be defined in the description of any object defined using this type.
- `TimeInterval`—Represents a period of time measured in units of 0.01 seconds.
- `DateAndTime`—Represents a date-time specification.

24.4 MIB Life Cycle

To write a MIB, you should be familiar with the phases in the life of a MIB:

1. Conception—The need for a MIB commonly results from engineering or marketing pressures for standardized, distributed management of a technology. At this stage, it is important to determine whether an existing standard or a Cisco-enterprise MIB already exists that might provide some or all of the management pieces. Your primary avenues of research are to examine IETF work and MIBs already implemented by Cisco and to consult with the Cisco MIB police.
2. Design—The actual design of a MIB is discussed in the section "Design a MIB" later in this chapter.
3. Implementation—Implementing a MIB is discussed in the section "Establish a New MIB" later in this chapter.
4. Release—Releasing a MIB is similar to releasing any other software, except that a MIB release also includes the MIB description file. For details on the MIB release process, see: http://wwwin-eng.cisco.com/Eng/IOS/SNMP_WWW/Mib-Release/index.html. Also, see the section "Release a MIB" later in this chapter.

   Part of the release process includes supplying converted MIBs in SNMPv1 or SunNet Manager schema form.
5. Maintenance—Maintenance of a MIB is primarily the responsibility of the original developer, group, or individual. See the section "Maintain a MIB" later in this chapter.
6. Death—When a MIB is superseded by another, its objects are deprecated as described in the SMI and eventually made obsolete. Removal of the MIB from code must follow normal Cisco procedures for backward compatibility.

24.5 Design a MIB

24.5.1 MIB Design: Overview

Like software design in general, designing MIBs is a combination of art and science. To do it well, you must consider the following:

- Objective rules—The written requirements for MIB syntax and components, as defined for SNMPv2, plus Cisco's requirements and conventions. Objective rules are discussed in the appropriate RFCs and in the section "Follow MIB Conventions" in this chapter.
- Subjective conventions—Ways that people tend to design MIBs. Following subjective conventions reduces overall confusion and the likelihood of breaking objective rules, but sometimes an unconventional idea is a good one. Subjective conventions are those that come to

matters of opinion about what might best suit users or application programs. Such opinions are often intuitive and based on experience with other MIBs. The textual names (descriptors) for MIB objects and some of the organization of the MIB itself are examples of subjective conventions. Devising a creative, new way to use MIB objects can be dangerous, however, because you may do something that is invalid or so different from normal practice that it will not work even with reasonable applications.

- Audience—Who the MIB and its description are for. MIBs are primarily for the people who manage network devices, secondarily for people who support network managers, and least of all for software developers. If some part of the MIB is addressed primarily to software developers, you should indicate this explicitly, to avoid overly concerning the people who use the MIB to manage network devices. The MIB description is primarily for the MIB implementer, to define correct operation of the MIB objects in an implementation-independent way. It is secondarily for users of the MIB, although it often becomes their primary documentation.
- Purpose—Justification for the MIB. The purpose of every MIB object must be clear and relevant to monitoring and controlling the network device.

24.5.2 SNMP Application Considerations

An SNMP application is software that uses SNMP to control or observe systems other than the managing system. Examples are gathering statistics, configuring a system, and observing current operation. SNMP applications can range from simple, command-line programs that retrieve individual MIB objects to massive, graphical management systems. You can develop SNMP applications in C, higher-level languages such as TCL/TK, or something in between, such as C++.

When designing SNMP applications, consider the following issues:

- Platform—Consider the type of system on which an application will run and what services the platform will provide. There are a few popular platforms, such as SunNet Manager, NetView/6000, and HP OpenView, but they do not have common programming interfaces to their services. You often need to decide whether an application should be self-sufficient or should include platform dependencies. The latter solution results in multiple versions of the application.
- User needs—User needs are generally unclear. Many users do not understand the technology to be managed or the technology used for management, and vendors commonly do not understand user problems.
- MIB design—The success of SNMP has resulted in a plethora of MIBs with a myriad of objects, few of them well documented or well understood. It falls upon you to provide objects of recognizable utility rather than supplying every bit of information imaginable or only what is easy. It is part of the SNMP philosophy that applications should be complex so that agents can remain simple. However, agents must supply a solid, useful base of information. When possible, design a MIB along with representative applications, but try to keep it from becoming overly application specific.

24.5.3 MIB Design Phases

The following are the phases in designing a MIB:

- Design the MIB Content
- Design the Notifications
- Design the MIB Organization

Designing the content of a MIB is by far the most difficult of the tasks.

When designing a MIB, you can use one of the following methodologies:

- Design a MIB by trial and error, shipping intermediate versions to see how they operate. You might be successful, but mistakes can be costly.
- Examine examples of well-designed MIBs. Newer Cisco MIBs such as the Ping MIB, the VINES MIB, and the Configuration Management MIBs are good examples. Most standard MIBs are also good examples.
- Get help from the Cisco Router Agent Software group, which offers in-house SNMP MIB consulting services.

24.5.3.1 Design the MIB Content

The basic SNMP philosophy is to provide MIB information that different applications can use in different ways and to put the computational burden on the application rather than on the agent or instrumentation. For example, MIB information is generally kept as counters that wrap and cannot be reset. Multiple applications can then sample the counters at different intervals for different purposes and perform computations on the raw data.

MIB information must have a clear purpose. If you make every conceivable status and counter visible or every parameter controllable, the volume of available data becomes overwhelming. Make sure that every object is associated with an understandable failure or observational need and that every settable parameter has clear, observable reasons for its values.

If you must consider asynchronous notifications such as traps, resolve the problems of reliability and flow control. For more information about asynchronous notifications, refer to the section "Implement SNMP Asynchronous Notifications" later in this chapter.

24.5.3.2 Design the Notifications

Starting with SNMPv2, SNMP defines the concept of *notifications*. All notifications have the same protocol format as a response message and contain a standard set of objects to which the MIB designer can add other objects. Notifications are defined in MIB modules with the `NOTIFICATION-TYPE` ASN.1 macro from the SMI.

From RFC 1905, which defines SNMP protocol messages and operations, the standardized contents of a notification are the following:

- `sysUpTime.0`—Timestamp indicating when the event occurred.
- `snmpTrapOID.0`—Unique identification for the notification, derived from `NOTIFICATION-TYPE`.

There are two types of notifications:

- Traps—Architecturally, traps are considered an agent-to-manager function. The agent sends them as unacknowledged datagrams, so it is possible for a trap message to disappear without a trace.
- Informs—Added in SNMPv2, architecturally, informs are considered a manager-to-manager function. A manager expects acknowledgment of a transmitted inform and can retransmit until an acknowledgment is received or the transmitting manager declares a failure.

The proper use of asynchronous notifications is one of the major points of controversy in SNMP and network management in general. Most of this controversy is due to misunderstanding, but some is due to honest disagreement. The controversy centers around the following areas:

- Polling Versus Alerts
- Reliable Delivery

- Information Flow Control

Polling Versus Alerts

Some amount of polling is necessary to determine the state of the network. For example, sometimes systems break in such a way that they cannot report, and all systems cannot diagnose themselves. However, constantly polling a large number of systems on a network requires a powerful polling engine and uses a lot of bandwidth.

Polling can be reduced by applying it intelligently and by distributing it. People tend to poll far too much, too often, in an inefficient way. This is somewhat the fault of management platforms, but is an area that must be improved. Distributing the polling can be done using mid-level managers. Work on this is under way in the IETF Distributed Management Working Group.

Asynchronous alerts have a place in the solution, but there are concerns regarding flow control.

Reliable Delivery

SNMP traps are unreliable. Do not put important information in traps and nowhere else. This is bad management design. Information that is put into traps must be available through some other means, such as a history table. If it is not important enough to warrant such means, it is not important enough to send as a trap.

Reliability has its limits. Reliability simply means that a sending system knows that a message has arrived at a destination. It does not mean the message has been understood or processed. Without extraordinary measures involving nonvolatile memory, no network communication can be more reliable than the network itself. If communication is lost and systems fail, information is lost.

Information Flow Control

Ultimately, the problem to resolve is intelligent control in times of network stress. The basic concept is that higher-level managing systems are more intelligent than managed systems and can better determine when to collect information. The concern is that unintelligent managed systems will flood the network and the managers with relatively useless notifications, particularly when the network and the managers are already under stress. This means that what is reported asynchronously must be chosen wisely and that the source must control the flow of notifications. It must be possible to disable notifications. In general, notifications should be disabled by default. Managed systems should offer algorithms or controls to keep the rate of notifications reasonable, but ultimately they must defer such choices to the managing system.

24.5.3.3 Design the MIB Organization

The most difficult MIB organizational design issue is table indexing. Sometimes the proper indexing is obvious, but it can easily become a tangle of trade-offs among search overhead, complexity, and prediction of what organization is most useful to applications. You must understand these issues in the context of how you expect the MIB to be used in order to develop an efficient organization.

24.5.4 Check for Existing MIB Implementations

Before implementing a MIB, determine whether an existing standard or a Cisco-enterprise MIB already exists that might provide some or all the management pieces. Examine MIBs from various sources, including the following:

- IETF standards—These are accepted Internet standards and do not change much. They are published as Requests for Comment (RFCs). They range from proposed to draft to full Internet standard. A proposed standard is most likely to change, a full standard is unlikely to change, and a draft is likely to change only in a backward-compatible way.
- IETF Internet drafts—IETF work in progress. Sometimes the best way to instrument technology is with an Internet draft MIB, which is typically being worked on by an IETF working group. These MIBs can be unstable, so you must capture the specific Internet draft and place the MIB within the Cisco enterprise MIB branch (not in the experimental branch).
- Cisco enterprise MIBs—Cisco enterprise MIBs add instrumentation not covered by standard MIBs. As of Cisco IOS Release 10.2, Cisco has old MIBs and new MIBs. The old MIBs are from older software versions and often have somewhat unconventional features. The new MIBs are gradually replacing the old ones and adding new instrumentation.
- MIBs from other companies—Non-Cisco proprietary. It is occasionally appropriate to implement a MIB defined by some other company, especially when implementing technology that they originated and instrumented. Using these MIBs has problems similar to using IETF drafts in that you must capture the version of the MIB definition. However, the MIB itself should remain wherever in the MIB space the originating company put it so that the MIB can easily support existing applications.

To determine what MIBs Cisco implements, look at:
http://www.cisco.com/public/mibs/README ftp://ftp.cisco.com/pub/mibs/README.
This file is the key to determining which MIBs are in which products.

24.5.5 Ensure MIB Compliance

Currently, Cisco implementations of standard MIBs are often read-only or have some objects or object groups missing because of security concerns and time pressure for implementation. Each developer is responsible for documenting `AGENT-CAPABILITIES` specifics as described in RFC 2580. Eventually these documents will be made available to customers to supply to their applications.

24.5.6 Follow MIB Conventions

Cisco MIBs scrupulously follow IETF MIB standards and conventions, as well as Cisco conventions. All new Cisco MIBs must be in the format specified by the SNMPv2 SMI. As a service to our customers, we also convert all MIBS to SNMPv1 form.

24.5.6.1 Assigned Numbers

MIB branches are identified with unique numbers. The Internet Assigned Numbers Authority (IANA) assigns branch numbers to private enterprises. The Cisco Assigned Numbers Authority (CANA) assigns branch numbers within the Cisco branch to Cisco developers.

Before a MIB is approved by the Cisco MIB police, it might have a number in the `ciscoExperiment` branch. After the MIB is approved, the CANA assigns it a number in the `ciscoMgmt` branch. The `ciscoMgmt` number must be in place before you release the MIB.

To obtain a number in the `ciscoExperiment` or `ciscoMgmt` branch, use the CANA Web page:

http://wwwin-eng.cisco.com/Eng/IOS/SNMP_WWW/cana.html

24.5.6.2 Conventions for Writing MIBs

This section discusses some of the standard and Cisco-specific SNMP conventions that you should follow when writing MIBs. The conventions discussed here are those that people generally ask the most questions about. This discussion is not a complete description of MIB conventions.

For a complete discussion of standard SNMP conventions, refer to the following RFCs:

- RFC 2578—*Structure of Management Information for Version 2 of the Simple Network management Protocol (SNMPv2)*
- RFC 2579—*Textual Conventions forVersion 2 of the Simple Network Management Protocol (SNMPv2)*
- RFC 2580—*Conformance Statements for Version 2 of the Simple Network Management Protocol (SNMPv2)*

Cisco MIB Nomenclature

The Cisco conventions for overall MIB layout and naming comprise the following definitions. The capitalization of the MIB-specific items in the template represents the expected capitalization in the actual module. Do not use acronyms unless they are very well known for the given technology. Instead, use entire words or truncations, such as *Notification* or *Notif*, rather than leaving out vowels (for example, do not use *Ntfctn*).

- *MODULE-NAME*—The MIB module name. This name consists of the prefix `CISCO-`, the module name itself, such as `VINES` or `CONFIG-MAN`, and the suffix `MIB`. The module name is used to form the name of the MIB file. For example, the `CONFIG-MAN MIB` file is named `CISCO-CONFIG-MAN-MIB.m` .
- *name*—The name of the MIB module without hyphens, such as *vines* or *configMan*.
- *imports*—Standard `IMPORTS` statement for symbols from other MIB documents.
- *definition*—ASN.1 definition appropriate to the surrounding ASN.1 macro.
- *n*—The Cisco MIB number as assigned by CANA
- *abbr*—A short acronym for the MIB name, in the interest of keeping object descriptors manageable, such as *cv* or *ccm* for the VINES and configuration management.
- *section*—A MIB section name, such as *General*, *IP*, and *Counters*.
- *conventions*—Textual conventions for this module, if any.
- *objectName*—A MIB object name, such as *Descr*, *Inde* , and *FramesSent*.
- *eventName*—A MIB event name, such as *ConnectionClosed* and *LinkUp*.
- *groupName*—A MIB conformance group name, such as *FixedLength* and *Objects*.

Cisco MIBTemplate

The following is the Cisco MIB template:

```
MODULE-NAME DEFINITIONS ::= BEGIN
        imports

ciscoNameMIB MODULE-IDENTITY
        definition
        ::= { ciscoMgmt n }

ciscoNameMIBObjects OBJECT IDENTIFIER ::=  { ciscoNameMIB 1 }

abbrSection        OBJECT IDENTIFIER ::=  { ciscoNameMIBObjects 1 }

-- Textual Conventions

        definitions

-- Section

abbrSectionObjectNamer OBJECT-TYPE
        definition
        ::={ abbrSection 1 }

-- Notifications

ciscoNameMIBNotificationPrefix OBJECT IDENTIFIER ::= { ciscoNameMIB 2 }
ciscoNameMIBNotifications OBJECT IDENTIFIER ::=
        { ciscoNameMIBNotificationPrefix 0 }

ciscoNameEventName NOTIFICATION-TYPE
        definition
        ::= { ciscoNameMIBNotifications 1 }

ciscoNameMIBConformance OBJECT IDENTIFIER ::= { ciscoNameMIB 3 }
ciscoNameMIBCompliances OBJECT IDENTIFIER ::= { ciscoNameMIBConformance 1 }
ciscoNameMIBGroups      OBJECT IDENTIFIER ::= { ciscoNameMIBConformance 2 }

-- Conformance

ciscoNameMIBCompliance MODULE-COMPLIANCE
        definition
        ::= { ciscoNameMIBCompliances 1 }

-- Units of Conformance

ciscoNameGroupNameGroup OBJECT-GROUP
        definition
        ::= { ciscoNameMIBGroups 1 }

ciscoNameGroupNameGroup NOTIFICATION-GROUP
        definition
        ::= { ciscoNameMIBGroups 2 }

END
```

Example: MIB in CiscoTemplate

The following is an example of a production Cisco MIB in the standard Cisco template. Most of the objects have been removed from the example to shorten it.

```
CISCO-CONFIG-MAN-MIB DEFINITIONS ::= BEGIN

IMPORTS
        MODULE-IDENTITY,
        OBJECT-TYPE,
        NOTIFICATION-TYPE,
        TimeTicks,
        Integer32,
        Counter32,
        IpAddress
                FROM SNMPv2-SMI
        MODULE-COMPLIANCE, OBJECT-GROUP
                FROM SNMPv2-CONF
        DisplayString,
        TEXTUAL-CONVENTION
                FROM SNMPv2-TC
        ciscoMgmt
                FROM CISCO-SMI;

ciscoConfigManMIB MODULE-IDENTITY
        LA        ST-UPDATED“9506130000Z”
        OR        GANIZATION“Cisco Systems, Inc.”
        CONTACT-INFO
                “       Cisco Systems
                        Customer Service

                Postal: 170 W Tasman Drive
                        San Jose, CA  95134
                        USA

                        Tel: +1 800 553-NETS

                E-mail: cs-snmp@cisco.com”
        DESCRIPTION
                “Configuration management MIB.”
        RE        VISION“9506130000Z”
        DESCRIPTION
                “Initial version of this MIB module.”
        ::= { ciscoMgmt 38 }

ciscoConfigManMIBObjects OBJECT IDENTIFIER ::= { ciscoConfigManMIB 1 }

cc        mHistoryOBJECT IDENTIFIER ::= { ciscoConfigManMIBObjects 1 }


-- Textual Conventions

HistoryEventMedium ::= TEXTUAL-CONVENTION
        ST        ATUScurrent
        DESCRIPTION
                “The source or destination of a configuration change, save, or copy.

                er             aseerasing destination (source only)
                ru           nninglive operational data
                comm     andSourcethe command source itself
                st           artupwhat the system will use next reboot
                lo             callocal NVRAM or flash
                netw       orkTftpnetwork host via Trivial File Transfer
                net        workRcpnetwork host via Remote Copy
                “
```

```
                SYNTAX  INTEGER { erase(1), commandSource(2), running(3),
                        startup(4), local(5),
                        networkTftp(6), networkRcp(7) }

-- Configuration History

ccmHistoryRunningLastChanged OBJECT-TYPE
        SY     NTAXTimeTicks
        MAX-ACCESS read-only
        ST     ATUScurrent
        DESCRIPTION
                "The value of sysUpTime when the running configuration
                was last changed.

                If the value of ccmHistoryRunningLastChanged is greater than
                ccmHistoryRunningLastSaved, the configuration has been
                changed but not saved."
        ::= { ccmHistory 1 }

ccmHistoryRunningLastSaved OBJECT-TYPE
        SY     NTAXTimeTicks
        MAX-ACCESS read-only
        ST     ATUScurrent
        DESCRIPTION
                "The value of sysUpTime when the running configuration
                was last saved (written).

                If the value of ccmHistoryRunningLastChanged is greater than
                ccmHistoryRunningLastSaved, the configuration has been
                changed but not saved.

                What constitutes a safe saving of the running
                configuration is a management policy issue beyond the
                scope of this MIB. For some installations, writing the
                running configuration to a terminal may be a way of
                capturing and saving it. Others may use local or
                remote storage. Thus ANY write is considered saving
                for the purposes of the MIB."
        ::= { ccmHistory 2 }



-- Notifications
** Note: Much of the MIB is removed from this section.

ciscoConfigManMIBNotificationPrefix OBJECT IDENTIFIER ::= { ciscoConfigManMIB 2 }
ciscoConfigManMIBNotifications OBJECT IDENTIFIER ::= {
ciscoConfigManMIBNotificationPrefix 0 }

ciscoConfigManEvent NOTIFICATION-TYPE
       OBJECTS { ccmHistoryEventCommandSource,
                 ccmHistoryEventConfigSource,
                 ccmHistoryEventConfigDestination }
        STATUS  current
        DESCRIPTION
                 "Notification of a configuration management event as
                 recorded in ccmHistoryEventTable."
        ::= { ciscoConfigManMIBNotifications 1 }

-- Conformance
** Note: Much of the MIB is removed from this section.

ciscoConfigManMIBConformance OBJECT IDENTIFIER ::= { ciscoConfigManMIB 3 }
ciscoConfigManMIBCompliances OBJECT IDENTIFIER ::= { ciscoConfigManMIBConformance 1 }
ci      scoConfigManMIBGroupsOBJECT IDENTIFIER ::= { ciscoConfigManMIBConformance 2 }
```

```
-- Compliance

ciscoConfigManMIBCompliance MODULE-COMPLIANCE
        STATUS current
        DESCRIPTION
                "The compliance statement for entities which implement
                the Cisco Configuration Management MIB"
        MO        DULE-- this module
                MANDATORY-GROUPS { ciscoConfigManHistoryGroup }
        ::= { ciscoConfigManMIBCompliances 1 }

-- Units of Conformance

ciscoConfigManHistoryGroup OBJECT-GROUP
        OBJECTS {
                ccmHistoryRunningLastChanged,
                ccmHistoryRunningLastSaved
        }
        STATUS current
        DESCRIPTION
                "Configuration history."
        ::= { ciscoConfigManMIBGroups 1 }

ciscoConfigManHistoryNotifyGroup NOTIFICATION-GROUP
        NOTIFICATIONS { ciscoConfigManEvent }
        STATUS current
        DESCRIPTION
            "Configuration history notifications."
        ::= { ciscoConfigManMIBGroups 2 }

END
```

AGENT-CAPABILITIES Object Identifier

For MIBs in the `ciscoMgmt` tree, the `AGENT-CAPABILITIES` number is their MIB number plus 20.

64-bit Counters

For 64-bit counters, follow the SNMPv2 1-hour principle, which is defined in RFC 2578. For SNMPv1 compatibility, follow the RFC 2233 example by including a fast-wrapping 32-bit counter for the low 32 bits only.

For standard MIB modules, the `Counter64` type can be used only if the information being modeled would wrap in less than one hour if the `Counter32` type was used instead.

Objects in NVRAM

When you write to NVRAM, you must write everything; you cannot update just a single parameter. Also, writing to NVRAM should be done explicitly and only by the user; it should not be done as a side effect of an SNMP `Set` operation (with one exception). This is because many users run with a configuration that is different than the one in NVRAM in order to test a new configuration, and other users run with a configuration that is loaded from a TFTP server because the configuration is too large to fit in NVRAM. Overwriting NVRAM in these cases is a bad thing.

An SNMP `Set` request should behave exactly like the command-line interface **configure** command; it should modify the running configuration only. NVRAM should be written only when the **copy running-config startup-config** command is issued or when a `Set` request is issued to the SNMP object `writeMem`. Therefore, it is up to the user (or network management application) to ensure that one of these is performed before writing NVRAM.

To save the configuration on a TFTP server, use the `writeNet` SNMP object.

Indexing History Tables

For time-based history tables, use a monotonically increasing integer as the index, keeping a window of as many entries as allowed. Let the index wrap. If the MIB allows, wrapping can flush existing entries. This makes the implementation easier. See the Cisco Configuration Management MIB Event History Table as an example.

24.5.7 MIB Compilers

24.5.7.1 Function of MIB Compilers

A MIB compiler parses the SNMP SMI subset of ASN.1. Most MIB compilers generate a summary of the MIB contents that is easy to understand and that is suitable to be read by another program. Some MIB compilers generate source code, for example in C. Most often, MIB compilers are used to determine whether a MIB module is syntactically correct so that it has a chance of compiling when the module is compiled by another compiler.

When you are including a MIB with your code, your MIB is compiled as part of the standard `make` process.You should compile the MIB yourself to ensure that it will compile successfully during the `make` process.

24.5.7.2 Available Compilers

The best, most complete, and most powerful MIB compiler is the SNMP Management Information Compiler Next Generation (SMICng), which was designed and implemented by Dave Perkins. The code for this compiler is in `/nfs/csc/smicng`. The `/doc` subdirectory contains a user manual in the file `smicug.txt`.

Another commonly used MIB compiler is `mos` , which is part of the ISODE package. Although mosy is available at Cisco, it is not as complete as SMICng.

24.5.7.3 Invoke the MIB Compiler

To verify that there are no syntax errors in a new or modified MIB, use the `update-mibs.pl` script, which is supported by the MIB release group. This script invokes the SMIC MIB compiler.

To invoke the `update-mibs.pl` script, follow these steps:

Step Login to a SunOS4.*x* or Solaris 2.*x* system.

Step Add the directory `/nfs/csc/mib-release/bin` to your shell's path variable.

Step If you have not done so already, create a MIB workspace:

```
mibco.pl [release]
```

Step Copy your new or modified MIBs into the `text-mibs` directory in your workspace.

Step Change directories into the root of your workspace.

Step Compile the MIB:

If you are testing changes to an SNMPv2-style MIB, run the following command:

```
update-mibs.pl
```

If you are testing changes to an SNMPv1-style MIB, run the following command:

```
update-mibs.pl -1
```

24.5.8 Agent Development

Cisco's SNMP agent is derived from code purchased from and supported by SNMP Research. Development of the agent itself is a task separate from the development of MIBs and is the responsibility of the Cisco router agent software group.

24.5.9 Cisco Internal MIB Design Support

Individual technology groups typically design and implement their own MIBs. The Cisco router agent software group is responsible for technical oversight of Cisco's SNMP agent and MIBs and consulting with other groups throughout the company. This group is assisted by the MIB police who help people understand and follow the Cisco MIB conventions. To contact the MIB police, use the e-mail alias `mib-police`.

The Cisco Assigned Numbers Authority (CANA) is responsible for assigning unique numbers to identify MIBs and certain other SNMP elements, such as machine identifications. To contact the CANA, use the e-mail alias `cana`.

24.6 MIB Development Process: Overview

To develop a MIB, you perform the following tasks:

- Establish a New MIB
- Observe Modularity
- Implement MIB Objects
- Implement SNMP Asynchronous Notifications
- Compile a MIB
- Test a MIB
- Release a MIB
- Maintain a MIB

24.7 Establish a New MIB

To establish a new MIB, follow these steps. These steps use the sample MIB *BOOJUM-MIB.m* .

Step Create a development tree.

Step Do a **make depend** or a **make dependancies**) from the `sys` directory. This make generates code for all previously defined MIBs.

Step Determine the top-level identifier for your MIB, which is the top-level object identifier from which all the objects in your MIB are rooted.

Cisco MIBs typically have a standard structure. Using the `CISCO-CONFIG-MAN-MIB.my` MIB as an example, this MIB has the following top-level identifiers:

```
ciscoConfigManMIB       ciscoConfigManMIBObjects

                        ciscoConfigManMIBNotificationPrefix

                        ciscoConfigManMIBConformance
```

`ciscoConfigManMIB` is the root for everything in the MIB, including ordinary objects, notification objects, and the conformance section.

`ciscoConfigManMIBObjects` is the root of all the "ordinary" objects in the MIB.

`ciscoConfigManMIBNotificationPrefix` is the root of the notification objects.

`ciscoConfigManMIBConformance` is the root of the conformance section.

If your MIB implementation will not generate notifications, you can use the ordinary objects root (in this example `ciscoConfigManMIBObjects`) as your top-level identifier. If your MIB implementation will generate notifications, use the root for everything in the MIB (in this example `ciscoConfigManMIB`) as the top-level identifier.

As an example of a typical case, suppose that you are providing SNMP support for the `boojum` subsystem. All the existing `.c` and `.h` files for this subsystem are in the directory `sys/boojum`. This is also where you want the files that will be generated by the MIB compiler in Ste p7 to end up.

Assume that the MIB for `boojum` support is `BOOJUM-MIB.m` . The names of all the files that will be generated by the MIB compiler must have a common substring in common, a substring we get to choose. For this example, let's use `boojummi` . Also, we need to find the top-level identifier in `BOOJUM-MIB.m` . Let's assume that it is `boojumObjects`.

Step Add the new MIB to the `sys/MIBS` directory:

(a) Check out the `sys/MIBS` directory from ClearCase.

(b) Place the file `BOOJUM-MIB.my` into the `sys/MIBS` directory.

(c) Issue a **cleartool mkelem** command to define the element to ClearCase.

Note Before you can issue a **cleartool mkelem** command, you must have the parent directory checked out in your ClearCase view.

Step Create a MIB compiler configuration file.

Each invocation of the MIB compiler should be controlled via a configuration file. Continuing the example, you would create the file `sys/boojum/sr_boojummib.cfg`, which contains the `-group` *`group-name`* directive:

```
-group boojumObjects
```

The `-group group-name` directive defines the group of MIB objects for which the MIB compiler should generate code. The group name is the top-level identifier (`boojumObjects`) identified in Ste p3. (Note that you can have multiple `-group` directions in the `.cfg` file, for instance if you wanted to include both the ordinary and notification objects.)

The name of the stamp file must also include the common substring chosen in Step 3. Doing this is how the substring is communicated to the MIB compiler. This means that the stamp file name should be `sr_common-substring.stamp`, or, in this example, `sr_boojummib.stamp`.

In addition, add any other options that you want to be passed to the `mibcomp.perl` script when the MIB is compiled.

After creating the file, issue a **cleartool mkelem** command to define the element to ClearCase.

Step Update the `sys/makemibs` file to add the dependency for the new MIB.

Invocation of the MIB compiler is specified in the file sys/makemibs using an implicit makefile rule. Continuing the example, you would add the following line to `sys/makemibs`:

```
../boojum/sr_boojummib.stamp: ../MIBS/BOOJUM-MIB.def
```

Step Generate user-modifiable source files.

Files produced by the MIB compiler fall into two categories, nonmodifiable and user-modifiable. Most engineers are concerned only with the nonmodifiable files, which are generated during the **make depend** process. When implementing a MIB for the first time, however, you must also produce the user-modifiable files. You do this explicitly through the `%.code` rule in the `makefile`. Continuing the example, you would issue the following command from the `sys` directory, which causes the MIB compiler to generate both the nonmodifiable and user-modifiable files:

```
make boojum/sr_boojummib.code
```

The following nonmodifiable files are generated:

```
sys/boojum/sr_boojummibdefs.h
sys/boojum/sr_boojummibpart.h
sys/boojum/sr_boojummibsupp.h
sys/boojum/sr_boojummibtype.h
sys/boojum/sr_boojummiboid.c
sys/boojum/sr_boojummib.stamp
```

The following user-modifiable files are generated if they do not already exist:

```
sys/boojum/sr_boojummib.c
sys/boojum/sr_boojummib.h
```

The following user-modifiable file is generated if you specify the `-userpart` switch and the file does not already exist:

```
sys/boojum/sr_boojummibuser.h
```

The following user-modifiable file is generated if you specify the `-snmpmibh` switch and the file does not already exist:

```
sys/boojum/sr_boojummib-mib.h
```

Step Add your user-modifiable files to the ClearCase repository.

In Step 7, you generated two to four user-modifiable files if they did not already exist. If these files already existed, the MIB compiler does not overwrite these files because they are modified when you implement the MIB. Because they are dynamic source files, you should make them elements in the ClearCase repository.

You should never modify the other six fixes generated in Step7. Therefore, you should not placed these files under ClearCase control.

Step Implement the MIB by adding code to your parameter files.

In the sample MIB, the parameter files are the `sys/boojum/sr_boojummib.c` and `sys/boojum/sr_boojummib.h` files. You must add code to `sys/boojum/sr_boojummib.c`, specifically to the system-dependent method routines. You do not have to add code to `sys/boojum/sr_boojummib.h`.

If a `sys/boojum/sr_boojummibuser.h` file was generated, this file should define the macros that cause the user-supplied fields to appear in the various structures.

If a `sys/boojum/sr_boojummib-mib.h` file was generated, modify it to remove all but the most essential OID-to-identifier translation entries, because these entries consume code space but provide little benefit. At a minimum, remove the following entries:

```
Leaf object entries
Table entries
Notification entries unless explicitly referenced by the code
Compliance and Conformance entries
```

Typically only the following entries remain:

```
Top-level group entries
Entry entries
```

Step 1 0 Add `.o` file rules to the appropriate `makefile` so that the `sr_boojummib.o` and `sr_boojummiboid.o` files are placed in the proper subsystems.

Step 1 1 Build the system and test the code.

Step 1 2 Commit the code.

Make sure you commit the MIB file in `sys/MIBS`, the `makemibs` file, the `boojum/sr_boojummib.cfg` configuration file, the `boojum/sr_boojummib.c` and `boojum/sr_boojummib.h` files, the `makefiles` that were altered in St ep10, and if applicable, the `boojum/sr_boojummibuser.h` and `sr_boojummib-mib.h` files.

24.8 Compile a MIB

To compile a MIB, use the `sys/scripts/mibcomp.perl` script, also referred to as `mibcomp` or the MIB compiler.

Before you compile a MIB, you must determine the following:

- Which MIB or MIBs to Compile
- Which Groups to Compile
- Where to Place Files Generated by the MIB Compiler

24.8.1 Which MIB or MIBs to Compile

Generally, you compile only one MIB, the MIB you have implemented. The MIB filename must consist of the ASN.1 module name of the MIB followed by a `.my` extension. The MIB file must be in the `sys/MIBS` directory of your development tree.

If you have two closely related MIBs, you can compile them together, producing code that implements both MIBs. In particular, if you are implementing a MIB that contains a table that uses the SNMPv2 `AUGMENTS` keyword, you must compile your MIB together with the MIB that contains the table being augmented.

24.8.2 Which Groups to Compile

MIBs generally contain a top-level ASN.1 identifier, followed by one or more group identifiers, with individual objects being defined on a per-group basis. In some MIBs, individual objects are defined directly beneath the top-level identifier. In most cases, you want to implement every object specified in the MIB. You indicate this by passing the top-level identifier to the MIB compiler

There are two exceptions. The first is that if you cannot implement any of the objects in a group, you can specify only those groups with objects you can implement. The second exception is that if the MIB contains groups that must be implemented in separate subsystems in the Cisco code base, you should invoke the MIB compiler separately for each subsystem.

24.8.3 Where to Place Files Generated by the MIB Compiler

The MIB compiler generates up to nine C source files. You must choose a directory into which the MIB compiler should place them. Whenever possible, especially for straightforward MIBs, such as protocol MIBs, place the files in the subsystem that corresponds to the protocol. For less straightforward cases, such as media-specific MIBs where an associated media-specific directory does not exist, you might need to place the generated files into the `sys/snmp` directory. It is also acceptable to create a new directory to hold a MIB implementation.

All the files generated by the MIB compiler have a generic form of `sr_{`*`id`*`}*.c` and `sr_{`*`id`*`}*.h`, where you specify a string for *`id`*. Choose a string that is descriptive, but yet as few characters as possible in order to keep the filenames as short as possible. The *`id`* is used in the code as part of some function names, so it must consist entirely of legal C identifier characters, preferably only letters. It is helpful to include the string `mib` at the end of the *`id`*.

24.8.4 Makefile Rules for Compiling MIBs

All invocations of the MIB compiler are controlled by `makefile` rules. All the rules that are specific to the MIB compiler are in `sys/makemibs`, which is a `makefile` that contains three externally available target rules and other internal rules. Although `sys/makemibs` is a `makefile`, you should never explicitly execute the `make` command on `sys/makemibs` directly. Instead, invoke it from `sys/makefile` when you perform a **make dependancies** or a **make *.code**. You do this because the files that `sys/makemibs` generates must be present when the dependencies rule attempts to create the `.D.*` files that it uses to build the file `sys/dependancies`. If the files have not been generated, erroneous dependencies are calculated for any existing files that reference the files to be created.

The following are the MIB dependencies rules in `sys/makefile`:

```
@$(MAKE) $(MAKEFLAG-J) -C obj-68-c7000 -f ../makemibs --no-print-directory depend
@$(MAKE) $(MAKEFLAG-J) -C obj-68-c7000 -f ../makemibs --no-print-directory mibfiles
```

The `depend` rule in `sys/makemibs` generates a `.D.*` file for each `*.my` file in the `sys/MIBS` directory. These files contain the dependencies for creating a `.def` file from the associated `.my` file. When all the `.D.*` files have been created, they are combined into a single `sys/mibdependencies` file.

The `mibfiles` rule in `sys/makemibs` generates all the MIB source files.

In addition to the above two rules, the following rule in `sys/makefile` causes an associated rule in `sys/makemibs` to be processed:

```
%.code:
        @$(MAKE) $(MAKEFLAG-J) -C obj-68-c7000 -f ../makemibs \
                  --no-print-directory $@
```

This third rule causes the MIB compiler to generate user-modifiable source code, which is typically done only when a new MIB is being implemented.

24.8.5 Invoke the MIB Compiler

To invoke the MIB compiler, use the `sys/scripts/mibcomp.perl` script, also referred to as `mibcomp` or the MIB compiler. This script has the following syntax. Tabl e24-1 explains the options. (Table 24-2 lists the options supported for compatibility with previous version of the `mibcomp` scripts. These options will eventually be phased out.) All builds are performed in the various `sys/obj*` directories. Therefore, any path names that appear in `makefiles` must take this into consideration.

mibcomp.perl *options mibs*

Table 24-1 MIB Compiler Script Options

Option	Description
-f *name*	Specifies the name of a mibcomp configuration file.
-codegen	Generates the user-modifiable files in addition to generating the nonmodifiable source files.
-postmosy *name*	Specifies the group in the MIB to compile. You can specify multiple **-group** options. Each option can list only one group.
-debug	Enables trivial debugging output.
-cache	Enables the -cache switch in postmosy. This switch generates trivial caching in the system-independent method routines.
-row_status	Enables the -row_status switch in postmosy. This switch generates code that implements the `RowStatus TEXTUAL-CONVENTION`.
-userpart	Enables the -userpart switch in postmosy. This switch places a macro invocation in each structure in the `sr_`*`id`*`type.h` file. By defining macros in this file, you can cause additional fields to be placed in the structures automatically generated by postmosy When used in conjunction with the -codegen option, the -userpart option generates an `sr_`*`id`*`user.h` file.
-snmpmibh	Enables the -snmpmibh switch in postmosy. When used in conjunction with the -codegen option, the -snmpmibh option generates an `sr_`*`id`*`-mib.h` file, which contains OID-to-textual-identifier mappings.
mibs	MIB definitions to be compiled.

Table 24-2 MIB Compiler Script Options Supported for Backwards Compatibility

Option	Description
-g *group ...*	Deprecated version of the -group option.
-s *stampfile*	Specifies the name of the stamp output file. This file defines the path to the directory where the MIB compile will place its output. It also defines the actual naming convention for the output files. Avoid using this option; using the -f option instead.

24.8.6 What the MIB Compiler Does

The `mibcomp.perl` MIB compiler script generates C source code files and header files for the MIB. The script is a wrapper around the postmosy program, which is provided by SNMP Research.

The input to `mibcomp.perl` is MIB definition files with names in the format `*.def`. These files are produced from the initial `*.my` MIB files by the mosy program, which is also supplied by SNMP Research.

`mibcomp.perl` then passes its input to postmosy, which processes the `*.def` files and produces C code suitable for building the agent code required to instrument the MIB. For more information about the mosy and postmosy programs, see the documentation available from SNMP Research.

24.8.7 Output from the MIB Compiler

The MIB compiler generates up to ten files. The files are named according to the definition in the **-f** *file* option you provided to the `mibcomp.perl` MIB compiler script (or in the obsolete **-s** *stampfile* option).

The MIB compiler always generates the following files. These filenames assume that you specified a **-f** option in the format `../my_directory/sr_id`mib.cfg when you invoked the `mibcomp.perl` script.

- `../my_directory/sr_id`miboid.c
- `../my_directory/sr_id`mibdefs.h
- `../my_directory/sr_id`mibpart.h
- `../my_directory/sr_id`mibsupp.h
- `../my_directory/sr_id`mibtype.h
- `../my_directory/sr_id`mib.stamp

If you invoked the MIB compiler with the -codegen option, it generates the following two files if they do not already exist:

- `../my_directory/sr_id`mib.c
- `../my_directory/sr_id`mib.h

If you invoked the MIB compiler with the -codegen and -userpart options, it generates the following two files if they do not already exist:

- `../my_directory/sr_id`-user.h

If you invoked the MIB compiler with the -codegen and -smpmibh options, it generates the following two files if they do not already exist:

- `../my_directory/sr_id`-mib.h

24.8.8 Compile a MIB: Examples

This section provides some examples of compiling a MIB. Note that while these examples show the `mibcomp.perl` script being invoked directly, typically this is done via a `makefile`.

Compile the DS1 MIB

The DS1 MIB illustrates a case of compiling a simple MIB. This MIB is specified in RFC 1406. Its ASN.1 module name is RFC1406-MIB. Hence, when you extract the contents of the MIB from the RFC, you place them in the file `sys/MIBS/RFC1406-MIB.m`. You then run this file through `mosy` to produce `sys/MIBS/RFC1406-MIB.def`. This MIB contains the following top-level identifier:

```
ds1 OBJECT IDENTIFIER ::= { transmission 18 }
```

Cisco implements objects from every group. Therefore, you can specify `ds1` as the only group for which code is to be generated. All the Cisco DS1 code is in the `sys/hes` directory, and `ds1mib` is a reasonable identifier to assign the generated code. Hence, to compile the DS1 MIB, you would create a configuration file `sys/hes/sr_ds1mib.cfg` containing the following:

```
-group ds1
```

You would then invoke `mibcomp.perl` as follows:

```
mibcomp.perl -f ../hes/sr_ds1mib.cfg `mibreq.perl ../MIBS/RFC1406-MIB.def`
```

Compile MIB-II

MIB-II, defined in RFC 1213, illustrates a special case of compiling a MIB. The ASN.1 module name of the MIB is RFC1213-MIB. Hence, when you extract the contents of the MIB from the RFC, you place them in the file `sys/MIBS/RFC1213-MIB.m`. You then run this file through `mosy` to produce `sys/MIBS/RFC1213-MIB.def`. MIB-II is a special case because, although it contains the top-level identifier `mib-2`, it also contains the following groups: `system`, `interfaces`, `at`, `ip`, `icmp`, `tcp`, `udp`, `egp`, and `snmp`. Because the Cisco IP code is not in the same directory as the Cisco TCP or Cisco SNMP code, you cannot compile all the groups at once.

We have decided to invoke the MIB compiler individually for each group. To do this, we create three distinct configuration files:

```
sys/ip/sr_ipmib2.cfg contains “-group ip”
sys/ip/sr_icmpmib2.cfg contains “-group icmp”
sys/snmp/sr_snmpmib2.cfg contains “-group snmp”
```

Then we invoke `mibcomp.perl` three times with the following commands, once for the IP portion, once for the ICMP portion, and a third time for the SNMP portion:

```
mibcomp.perl -f ../ip/sr_ipmib2.cfg `mibreq.perl ../MIBS/RFC1213-MIB.def`
mibcomp.perl -f ../ip/sr_icmpmib2.cfg `mibreq.perl ../MIBS/RFC1213-MIB.def`
mibcomp.perl -f ../snmp/sr_snmpmib2.cfg `mibreq.perl ../MIBS/RFC1213-MIB.def`
```

Because the IP and ICMP code is in the same directory—`sys/ip`—you could compile these two groups together with a configuration file and command similar to the following. However, if you are going to compile some groups separately, it is cleaner to compile all the groups separately.

```
sys/ip/sr_ipicmpmib.cfg contains “-group ip -group icmp”
mibcomp.perl -f ../ip/sr_ipicmpmib2.cfg `mibreq.perl ../MIBS/RFC1213-MIB.def`
```

Compile the SNMPv2 MIB and SNMPv2 Party MIBs

RFC 2012 defines the TCP MIB, and the Cisco enterprise-specific TCP MIB provides some extensions to this standard MIB. Based on their ASN.1 module names, these MIBs are stored in the files `sys/MIBS/TCP-MIB.my` and `sys/MIBS/CISCO-TCP-MIB.m` . The `CISCO-TCP-MIB.my` file contains the following definition:

```
ci  scoTcpConnEntryOBJECT-TYPE
SY     NTAXCiscoTcpConnEntry
MAX-ACCESS not-accessible
ST     ATUScurrent
DESCRIPTION
“Additional information about a particular current TCP
connection beyond that provided by the TCP-MIB tcpConnEntry.
An object of this type is transient, in that it ceases to
exist when (or soon after) the connection makes the transition
to the CLOSED state.”
AUGMENTS { tcpConnEntry }
::= { ciscoTcpConnTable 1 }
```

Because `CISCO-TCP-MIB` has a table that `AUGMENTS` a table in the TCP-MIB, you must compile these two MIBs together. The Cisco TCP MIB has a top-level identifier of `ciscoTcpMIB` and the TCP MIB has a top-level identifier of `tcp`, so you would create the following configuration file:

```
sys/snmp/sr_tcpmib2.cfg contains “-group ciscoTcpMIB -group tcp”
```

You would then invoke `mibcomp.perl` as follows:

```
mibcomp.perl -f ../snmp/sr_tcpmib2.cfg \
`mibreq.perl ../MIBS/CISCO-TCP-MIB.def ../MIBS/TCP-MIB.def`
```

It is not necessary to specify `TCP-MIB` on the `mibreq.perl` command line, because the Cisco TCP MIB imports the TCP MIB. `mibreq.perl` determines that `TCP-MIB` is required. However, explicitly specifying both MIBs makes it clear that you are generating code for definitions in both MIBs.

24.9 Observe Modularity

You must observe modularity rules when dealing with SNMP subsystems and instrumentation.

24.9.1 Subsystem

SNMP is a separate subsystem that can be omitted from a system. All MIB implementations must observe this same modularity. They might have dependencies on SNMP, but nothing can depend on them except through a proper registry so that they can be omitted with SNMP.

24.9.2 Instrumentation

Instrumentation must be distinct and separate from the MIB code itself. For example, the counters in an Ethernet driver belong to the driver, not to SNMP. An Ethernet MIB needs access to them, but must do so through some interface that depends on the driver, whether that is global variables or preferably a set of interface procedures. Think of the MIB code as a translator that stands between the SNMP agent and the real data, keeping them distinct and separate. Even if the instrumentation is created only for SNMP availability, it must be separate. An example of this is the configuration management MIB.

24.10 Implement MIB Objects

This section discusses the following tips for creating SNMPv2 MIB method routines:

- GCC Warnings
- Validation
- k_get Routines
- k_set Routines

24.10.1 GCC Warnings

The code that is output by the MIB compiler causes GCC to generate many warnings. You can eliminate most of these warnings by doing the following:

- Change all function declarations to ANSI form.
- Preinitialize `*`data to `NULL` in all `get` functions. This eliminates the warning "data possibly used without being set."
- Delete all unused variable declarations.

24.10.2 Validation

Your `test` routines should perform sufficient testing so that when the `set` routines are called, they should not fail. Testing should include consistency checks among the objects in a row. If you are adding objects to the `dp->data` but you do not have all the objects required to fully create or modify a row, or if you have modified a field such that it becomes inconsistent with another field, set `dp->state` to `UNKNOWN`. This settings tells the SNMP engine that the given value was legal, but that the row currently is not legal.

In your `test` routines, you should almost always insert code in the `case` statement that performs some kind of validation. For example, the MIB compiler generated the following for MIB-II IP group:

```
switch (object->nominator) {

#ifdef I_ipForwarding
    case I_ipForwarding:

        SET_VALID(I_ipForwarding, ((ip_t *) (dp->data))->valid);

        ((ip_t *) (dp->data))->ipForwarding = value->sl_value;
        break;
#endif /* I_ipForwarding */
```

Validation was added to this code to produce the following:

```
switch (object->nominator) {

#ifdef I_ipForwarding
    case I_ipForwarding:

        if ((value->sl_value != D_ipForwarding_forwarding) &&
             (value->sl_value != D_ipForwarding_not_forwarding))
            return(WRONG_VALUE_ERROR);

        if (!router_enable)
            return(INCONSISTENT_VALUE_ERROR);

        SET_VALID(I_ipForwarding, ((ip_t *) (dp->data))->valid);

        ((ip_t *) (dp->data))->ipForwarding = value->sl_value;
            break;
#endif /* I_ipForwarding */
```

24.10.3 k_get Routines

`k_get` routines are generally fairly straightforward to create. For scalar items, you just need to get the items. For tabular items, you need to scan the associated table until you find the correct entry, and then copy the data into the holding area.

24.10.4 k_set Routines

`k_set` routines are generally not so straightforward to create. The `k_set` stubs are not much help. In order to craft these routines, start by referencing back to the associated `test` routine to see which items are settable. To do this for scalar objects, you add code similar to the following for each item:

```
if (VALID(I_{objectname}, data->valid)) {
    set object based on data->{objectname}
}
```

For table objects, there are three possibilities:

- When updating an existing row, you should locate the appropriate entry using the index objects.
- When creating a new row, you should acquire an empty row using the index objects by whatever method is appropriate for the given table. You can use code similar to that shown for scalar objects to fill in the row.
- When deleting a row, you use the index objects to locate the row and then delete it by whatever method is appropriate for the table). You can use code similar to that shown for scalar objects to fill in the row.

24.11 Implement SNMP Asynchronous Notifications

SNMP has two types of asynchronous notifications, traps and informs. Traps are unacknowledged datagrams. Informs are acknowledged datagrams sent from one manager process to another.

Implementing SNMP notifications is relatively straightforward, although not particularly simple. Before attempting to implement SNMP notifications, you should be familiar with SNMP and the Cisco development environment.

To implement SNMP notifications, you need to do the following:

- Decide Where to Place SNMP Notification Code
- Define the Notification
- Control the Notification
- Generate the Notification

24.11.1 Decide Where to Place SNMP Notification Code

Place the code for SNMP notifications in a "modularly appropriate place." Finding such a place is not always obvious; you need to consider both basic modularity and Cisco IOS subsystems when identifying a location. The typical MIB implementation is concerned with three subsystems: the SNMP subsystem, the MIB module subsystem, and the subsystem (or subsystems) with the instrumentation. For the most part, the code to support notifications should be in the MIB module subsystem, which can then make direct calls into the SNMP subsystem. The instrumentation subsystems make registry calls to declare events to the MIB module subsystem.

24.11.2 Define the Notification

Notifications are defined in SNMPv2 MIBs with the `NOTIFICATION-TYPE` macro. A good way to design notifications is to look at an example. Remember that at best traps are an optimization. They do not eliminate the need for polling or for providing the information in some other way.

Notifications automatically include a timestamp, so you do not need to provide code to do this. You also do not need instance objects as long as you include any object from a table, because the instance values are embedded in the OID of every object in the table.

Use notifications with care, because they can easily cause traffic problems on the network. Furthermore, traps are not reliable. If the information in the trap is important, make it available through some other means, such as an event history table. An example of a notification design using an event history table is in the Cisco Configuration Management MIB. The relevant parts of that design and implementation can be found in the development tree in the following files:

```
MIBS/CISCO-CONFIG-MAN-MIB.my
snmp/sr_configmanmib.c
snmp/config_history.c
```

24.11.3 Control the Notification

You need to specify a way to turn notifications on and off. At a minimum, you need a pair of command lines that affect all the notifications from a particular module or MIB. Some MIBs have an individual switch object for each of their notifications. It is debatable as to whether this is the correct model. A preferable model might have a central MIB for notification control that works across all MIBs rather than having to sort through all the MIBs to find many individual controls. However, such a standard or Cisco-proprietary MIB does not exist.

The following two commands control notifications. The older one is the **snmp-server host** command:

snmp-server host *host community-string* [*family* ...]
no snmp-server host *hostname*

This command configures trap receivers, and optionally places limits on the types of traps that can be sent to those receivers. If no family is specified, all traps are sent to the specified host. This command does not enable or disable the traps themselves.

The newer command is the **snmp-server enable** command:

[**no**] **snmp-server enable** {**traps** | **informs**} [*family...*]

This command enables or disables generation of a family of notifications. As of this writing, the portion of this command for informs is not yet implemented.

To implement control of a new notification family, make the code changes described in the following steps. In all the examples in these steps, `snark` is the family name and `boojum` is the individual notification name.

Step In the `parser/parser_defs_snmp.h` file, add `TRAP_ENABLE_SNARK`, `TRAP_SNARK`, `SNMPTRAPID_SNARK`, and `SNMPTRAPSTR_SNARK`, following the conventions already there.

Step Add `#includes` similar to the following for the parser:

```
#include “config.h”
#include “parser.h”
#include “../parser/actions.h”
#include “../parser/macros.h”
#include “../parser/parser_defs_parser.h”
#include “../parser/parser_defs_exec.h”
```

Step Add your option to the **snmp-server host** command:

- Add the parse chain, along with any other additional parse chains for your module, to `snark_chain.c` or in some other modularly appropriate place:

```
LINK_EXIT(cfg_snmp_host_snark_exit, no_alt);
KEYWORD_OR(cfg_snmp_host_snark, cfg_snmp_host_snark_exit, NONE,
    OBJ(int,1), (1<<TRAP_snark), “snark”, “Allow SNMP snark traps”, PRIV_CONF);
LINK_POINT(cfg_snmp_host_snark_entry, cfg_snmp_host_snark);
```

- In a modularly appropriate place, add the following to an existing `parser_extension_request` array or as a new one:

```
{ PARSE_ADD_CFG_SNMP_HOST_CMD, &pname(cfg_snmp_host_snark_entry) },
{ PARSE_ADD_CFG_SNMP_HOST_EXIT, (dynamic_transition *)
    &pname(cfg_snmp_host_snark_exit) },
```

To create your own new extension structure, bracket these lines with the following:

```
const parser_extension_request snark_chain_init_table[] = {
    your lines
    { PARSE_LIST_END, NULL }
};
```

Also, make the following call:

```
parser_add_command_list(snark_chain_init_table, “snark”);
```

- Add the function to generate a command line to save your configuration in some modularly appropriate place:

```
void config_history_snmp_host_nvgen (ulong flags)
{
    nv_add(flags & (1 << TRAP_SNARK), “ snark”);
}
```

- Add the following function to the registry in some modularly appropriate place at initialization time:

```
reg_add_snmp_host_nvgen(snark_snmp_host_nvgen, “snark_snmp_host_nvgen”);
```

Step Add your option to the **snmp-server enable** command:

- Add the parse chain, along with any other additional parse chains for your module, to *snark_chain.c* or in some other modularly appropriate place:

```
LINK_EXIT(cfg_snmp_enable_snark_exit, no_alt);
KEYWORD_OR(conf_snmp_enable_snark, cfg_snmp_enable_snark_exit, NONE, OBJ(int,1),
    (1<<TRAP_ENABLE_snark), “snark”, “Enable SNMP snark traps”, PRIV_CONF);
LINK_POINT(cfg_snmp_enable_snark_entry, conf_snmp_enable_snark);
```

- Add the following to an existing parser_extension_request array or as a new one in a modularly appropriate place:

```
{ PARSE_ADD_CFG_SNMP_ENABLE_CMD, &pname(cfg_snmp_enable_snark_entry) },
{ PARSE_ADD_CFG_SNMP_ENABLE_EXIT,
    (dynamic_transition *) &pname(cfg_snmp_enable_snark_exit) },
```

- In a modularly appropriate place, add the function to set your control variable (called from your parse chain):

```
void snark_trap_cfg_set (boolean enable, ushort flags)
{
    if ((flags & (1 << TRAP_ENABLE_SNARK))) {
        snark_enabled = enable;
    }
}
```

- In a modularly appropriate place, add the function to generate a command line to save your configuration:

```
void snark_trap_cfg_nvwrt (parseinfo *csb)
{
    nv_write(snark_traps_enabled, “%s traps snark”, csb->nv_command);
}
```

- In a modularly appropriate place, add the functions to the registry at initialization time:

```
reg_add_Trap_cfg_set(snark_trap_cfg_set, “snark_trap_cfg_set”);
reg_add_Trap_cfg_nvwrt(snark_trap_cfg_nvwrt, “snark_trap_cfg_nvwrt”);
```

Step Identify to SNMP which family the notification is in. In some modularly appropriate place, at initialization time, add functions to tell SNMP about the notification:

```
static const OID boojumOID = {LNboojum, (ulong *)IDboojum};
static char boojumOID_str[80];
MakeDotFromOID((OID *)&boojumOID, boojumOID_str);
register_snmp_trap(TRAP_snark, boojumOID_str);
```

The character string `boojumOID_str` is needed to generate the notification, which is why it is defined as `static`.

24.11.4 Generate the Notification

Outside of SNMP, the instrumented subsystem declares an event, most likely with a registry call to a notification-specific procedure in the related SNMP MIB module. Using input parameters from the instrumented subsystem, the notification procedure builds a list of objects to include in the notification and passes the event to SNMP, which is responsible for collecting the objects, building the message, and sending it.

To generate the notification, make the code changes and take the actions described in the following steps. In all examples, `boojum` is the name of the notification.

Have the instrumented subsystems declare the event:

Step Define the following service in the registry for the instrumented subsystem:

```
DEFINE boojum
/*
 * This service should be called when boojum happens.
 */
LIST
    void
    { input parameter declarations }
END
```

Step Include the following registry in the related MIB module and in the instrumented subsystem. *`name`* is the registry name.

```
#include "../subdirectory/name_registry.h"
```

Step Register the service in the related MIB module:

```
reg_add_boojum(boojum, "boojum");
```

Step In the instrumented subsystem, call the event declaration at the appropriate time. A typical input parameter might be a selector for a table entry.

```
reg_invoke_boojum(input_parameters);
```

Set things up so that SNMP can generate the notification. This example is based on the Configuration Management MIB, which has one notification with three objects, all from the same table with a single, integer index.

Step Define some constants.

`boojumTrapOID` and `boojumTrapOID_str` were defined above in the context of telling SNMP that the trap exists.

Define the individual notification number from the tail end of `NOTIFICATION-TYPE`.:

```
#define BOOJUM_NUMBER 1
```

Define the number of objects in the notification:

```
#define BOOJUM_VARBIND_COUNT 3
```

The OID as used for SNMPv1 traps:

```
static const OID enterpriseOID =
{LNboojum - 2, (ulong *)IDboojum};
```

Make a list of the objects in the trap:

```
static const OID boojum_varbinds[BOOJUM_VARBIND_COUNT] = {
    {LNobject1, (ulong *)IDobject1},
    {LNobject2, (ulong *)IDobject2},
    {LNobject3, (ulong *)IDobject3}
};
```

The `LN` and `ID` identifiers come from `sr_???mibpart.h`, which is `#included` with `sr_xxx.h`.

Step Set up the procedure and temporary variables:

```
void
boojum(int index)
{
    ulong       instance[1];
    int         i;
    OID         *vbList[BOOJUM_VARBIND_COUNT+1];
    OID         instanceOID;
}
```

Step Ensure that the family is enabled:

```
if (!boojum_traps_enabled)
    return;
```

Step Build an instance vector:

```
instance[0] = index;
instanceOID.oid_ptr = instance;
instanceOID.length = 1;
```

Step Build the real, NULL-terminated list of objects by OID, including instances:

```
for (i = 0; i < BOOJUM_VARBIND_COUNT; i++) {
    vbList[i] = CatOID((OID *) &boojum_varbinds[i], &instanceOID);
}
vbList[i] = NULL;
```

Step Have SNMP generate the trap:

```
snmp_trap(ENTERPRISE_TRAP, BOOJUM_NUMBER, vbList, (OID *)&enterpriseOID,
boojumOID_str);
```

Step Clean up after `CatOID`:

```
for (i = 0; i < TRAP_VARBIND_COUNT; i++) {
    FreeOID(vbList[i]);
}
```

24.12 Test a MIB

When testing a MIB, you check the following basic SNMP operations on objects and notifications:

- `Get`
- `GetNext`
- `Set`
- `Trap`

24.12.1 Test an Object

For each object in the MIB, test the following:

- Check that all objects—including scalars and first, intermediate, and last table entry—can be retrieved with `Get` and `GetNext` asking for an exact OID.
- Check that `Get` and `GetNext` work correctly in all situations of asking for a nonexistent OID, including before and after the MIB, before and after existing objects, OIDs that are too long (including variants of indexes in the instance portion that are too long), truncating subidentifiers from the right when using `GetNext`, and tables that are empty, partly filled, and full.
- Check that `Sets` work with valid values and fail properly with invalid values.
- Check that interdependent `Sets` work correctly when in the same request or separate requests and in any order.
- Confirm that tables fill correctly and behave correctly on overflow, and that the create and delete the first, intermediate, last, and overflow entries.
- Confirm that each object works as specified in the MIB, that a `Get` returns an operationally correct value, and that a `Set` has an operationally correct effect.

24.12.2 Test a Notification

For each notification, test the following:

- It is generated at the correct time with the correct information.
- It is enabled and disabled correctly via command line or MIB object.
- The correct control commands appear in system configuration.

24.12.3 Tools for Testing a MIB

To test a MIB, you can use command-line tools, X Windows tools, and notification tools.

24.12.3.1 Command-Line Tools

You can use the following command-line applications to test MIB objects:

- `getone`—Uses `Get` to obtain the value of one or more objects.
- `getnext`—Uses `GetNext` to obtain the value of one or more objects.
- `getmany`—Uses `GetNext` to obtain the values of a group of objects starting from a specified point in the MIB tree.
- `setany`—Uses `Set` to change the value of one or more objects.

Using these applications is relatively straightforward. Refer to the online `man` pages for details.

If you are working on a released MIB or know the translations, these applications can use the text form for OIDs and enumerations rather than just numbers. While you are developing a MIB, you can supply your own translations as follows:

Step Create a directory.

Step Copy all the files from `/nfs/csc/snmpv2/mibs` into the directory you created.

Step In the new directory, change the `makefile` as follows:

- Add your new MIB.

Step Do a `make` in that directory.

Step Set the environment variable `SR_MGR_CONF_DIR` to point at the new directory.

24.12.3.2 X Windows Tools

In an X Windows environment, you can use `xsnmptcl`, which is a TCL/TK application that has a number of built-in tests.

`xsnmptcl`, like all TCL tools, uses a lot of CPU, so you should run in on a nonmail server that has spare CPU cycles.

To run `xsnmptcl` the test suite, follow these steps:

Step Change into the `xsnmptcl` directory:

```
cd /atm1/kzm/isode-8.0/snmpV2/tcl/library
```

Step Start `xsnmptcl` for testing:

```
../xsnmptcl -f testing.tcl
```

The main window appears .

Step Use the buttons and dialog boxes on the main window to set up your test system as a context with a server name and community string.

Step Left double-click a test to start it. Right double-click for a description of the test.

If you start `xsnmptcl` with the following command, a variety of SNMP tools is also made available:

```
../xsnmptcl -f everything.tcl
```

To supply `xsnmptcl` with the number to text mappings for your MIB, issue the following commands:

```
cp sdurham/public/objects.defs yourdirectory/yourobjects.defs
/atm1/kzm/isode-8.0/snmpV2/mosy/xmosy yourrmib.my
cat /xxx/objects.defs yourmib.defs > yourobjects.defs
../xsnmptcl -o objects.defs -f testing.tcl
```

Do not change the standard `objects.defs` file. Instead, make a own copy in your own area.

24.12.3.3 Notification Tools

To test your notification, you need to cause it to happen. To check whether the notification is being handled correctly, you can use the **tcpdump** command from a SPARC system:

tcpdump -s 484 host *routername* **and port** *snmp-trap*

routername is the IP address of the router sending the notification. The **tcpdump** command first displays an error message about MIBs, then it waits for traps to arrive. It displays the objects in the trap by OID, type, and value. However, **tcpdump** does not interpret the trap header correctly. It will probably think your trap has something to do with X.25.

Remember that you must enable your traps and set up the SPARCas a trap host using the command options you just implemented.

To confirm that your test setup works, enable link-state traps on an interface, then shut it down:

```
config term
interface eth2
snmp trap link-status
shutdown
```

24.12.4 SNMP Operations

The following basic SNMP operations are the same for any MIB:

- `Get`
- `GetNext`
- `Set`
- `Trap`

For each object in the MIB, confirm the following functions:

- All objects can be retrieved with `Get` and `GetNext` asking for an exactly correct OID, including scalars and first, intermediate, and last table entry.
- `Get` and `GetNext` work correctly in all situations of asking for a nonexistent OID, including before and after the MIB, before and after existing objects, OIDs that are too long (including too-long variants of indexes in the instance portion), truncating subidentifiers from the right when using `GetNext`, and tables that are empty, partly filled, and full.
- `Sets` work with valid values and fail properly with invalid values.
- Interdependent `Sets` work correctly when in the same request or separate requests and in any order.

Confirm the following table operation:

- Fill correctly and behave correctly on overflow.
- Created and delete first, intermediate, last, and overflow entry

24.12.5 Object Functions

Confirm that each object works as specified in the MIB:

- `Get` returns an operationally correct value.
- `Set` has an operationally correct effect.

24.13 Release a MIB

24.13.1 Release MIB Code

Releasing MIB code is no different than releasing any other code; it requires testing and code review. At least one of the code reviewers should be an experienced MIB implementer. A good place to look for such a reviewer is the e-mail alias `mib-police`.

24.13.2 Release MIB Files

The MIB description files are released to customers along with the running code, and are an important part of the release. They are released via CCO. The procedure for putting them in the right place and generating SNMPv1 and SunNet Manager conversions has not been made public for general use. To initiate this process, contact the e-mail alias `mib-release`.

24.14 Maintain a MIB

As with other code, MIB code requires bug fixes and is not exempt from the requirements of backward compatibility. Once released, the semantics and naming of a MIB object are not formally allowed to change. Although practicality leads to breaking this rule occasionally, in general you must observe it.

To change a released MIB object significantly, you must remove the old object and add a new one.

To remove a released MIB object, you change its status to deprecated to indicate that it is going away, but leave it in the code. In a future release, you change the object to obsolete to indicate that it is gone, and remove it from the code. At this point, you can remove details of its description; however, its descriptor and OID remain reserved.

To add MIB objects, append them in an appropriate place in the appropriate MIB module.

When you add or remove MIB objects, you must update the compliance groups at the end of the MIB module. You cannot change existing compliance groups, but rather must always add new ones to reflect the changes. One way to handle MIB versions is to use compliance groups with `AGENT-CAPABILITIES` files.

The most typical modification to an existing MIB is to add new enumerations to an enumerated object. This type of change has no impact on any of the MIB sources that have been committed to the release tree. However, if an object is added, removed, or has its syntax modified, the resulting generated code will probably no longer be compatible with the `sr_`*`xxx`*`mib.c` code that was previously created. In this case, the `sr_`*`xxx`*`mib.stamp` file contains the skeleton code for the new MIB. It is up to you to compare the existing `sr_`*`xxx`*`mib.c` code to the new skeleton code in `sr_`*`xxx`*`mib.stamp` and to make all appropriate changes to the .c file before committing the MIB (and the `.c` file) to the release tree.

24.14.1 Use MIB Versions

SNMPv2 SMI provides version control with the following features, as specified in RFC 2578:

- The `MODULE-IDENTITY` macro can have numerous `REVISION` clauses that specify the changes to the MIB.
- Added, changed, or removed objects introduce new `OBJECT-GROUPS` and ne `MODULE-COMPLIANCE` statements.

MIB compliance is specified by listing `OBJECT-GROUPS` in an `AGENT-CAPABILITIES` file. `MODULE-COMPLIANCE` statements in this file explicitly define the compliance.

The following hypothetical example illustrates the use of compliance groups. This example shows the CISCO-ENVMON-MIB MIB, which has the following compliance statement:

```
ciscoEnvMonMIBCompliance MODULE-COMPLIANCE
        STATUS  current
        DESCRIPTION
                "The compliance statement for entities which implement
                the Cisco environmental monitor MIB"
        MODULE  -- this module
                MANDATORY-GROUPS { ciscoEnvMonMIBGroup }
        ::= { ciscoEnvMonMIBCompliances 1 }
```

ciscoEnvMonMIBGroup is defined as follows:

```
ciscoEnvMonMIBGroup OBJECT-GROUP
        OBJECTS {
        [..list deleted for brevity..]
        }
        STATUS  current
        DESCRIPTION
                "A collection of objects providing environmental monitoring
                capability to a Cisco chassis."
        ::= { ciscoEnvMonMIBGroups 1 }
```

The CISCO-ENVMON-MIB MIB would declare compliance with the ciscoEnvMonMIBCompliance compliance statement.

To add an object to this MIB that reports the humidity in the chassis, you would do the following:

Step Add a REVISION clause to the MODULE-IDENTITY macro explaining the update.

Step Update the LAST-UPDATED clause in the MODULE-IDENTITY macro.

Step Add an invocation of the OBJECT-TYPE macro to define the object:

```
ciscoEnvMonHumidity OBJECT-TYPE
        SYNTAX     Integer32 (0..100)
        MAX-ACCESS read-only
        STATUS     current
        DESCRIPTION
                "The relative humidity of the air in the managed device,
                measured as a percent of 100."
        ::= { ciscoEnvMonObjects xx }
```

Step Add an invocation of the OBJECT-GROUP macro that describes the added object:

```
ciscoEnvMonHumidityMIBGroup OBJECT-GROUP
        OBJECTS {
                ciscoEnvMonHumidity
        }
        STATUS  current
        DESCRIPTION
                "A collection of objects providing humidity monitoring
                capability to a Cisco chassis."
        ::= { ciscoEnvMonMIBGroups 2 }
```

Step Add an invocation of the MODULE-COMPLIANCE macro that describes the new maximum level of compliance:

```
ciscoEnvMonMIBComplianceRev1 MODULE-COMPLIANCE
        STATUS  current
        DESCRIPTION
                "The compliance statement for entities which implement
                the Cisco environmental monitor MIB"
        MODULE  -- this module
                MANDATORY-GROUPS {
                        ciscoEnvMonMIBGroup,
                        ciscoEnvMonHumidityMIBGroup
        }
        ::= { ciscoEnvMonMIBCompliances 2 }
```

Step Write AGENT-CAPABILITIES that describe the software release at which compliance with the new MIB occurred.

An implementation of the revised MIB could claim conformance with either the ciscoEnvMonMIBCompliance compliance statement if it does not support the humidity object or the ciscoEnvMonMIBComplianceRev1 compliance statement if it does support the humidity object.

Instead of defining the new OBJECT-GROUP with just the humidity object, you could define one with all the objects. Then the new MODULE-COMPLIANCE would specify only the new object group in its MANDATORY-GROUPS clause.

To delete a MIB object, follow these stepxs:

Step Update MODULE-IDENTIT .

Step Update the STATUS clause of the OBJECT-TYPE macros of the objects to obsolete. You never remove objects from a MIB because one or more OBJECT-GROUP macros might reference it. An object to be deleted should usually go through a time when its STATUS is deprecated to indicate that it will be removed. During this period, it remains implemented but any existing usage should stop.

Step Create a new OBJECT-GROUP macro that does not contain the deleted objects.

Step Create a new MODULE-COMPLIANCE macro that references the new object group macro.

24.15 Testing and Publishing a MIB

This section, formerly *Cisco IOS Technical Note 2: Testing and Publishing a MIB* (30 October 1996, ENG-9867), describes how to test a Simple Network Management Protocol (SNMP) Management Information Base (MIB) that you have developed and then publish it so that it is accessible to customers.

24.16 Create or Update a MIB Workspace

Before you can test a MIB, you must create or update a MIB *workspace*, which is your personal copy of the MIB repository. As with the Concurrent Versions System (CVS) or ClearCase, you have a private directory where you can access files from the repository. You then perform your builds, compiles, tests, and other operations from within that workspace.

To create a new workspace or update an existing workspace, use the mibco.pl script, which does the following:

- Creates the directory structure of a MIB workspace if you are creating a new workspace

- Creates a `MIBS` subdirectory in the MIB workspace in which you can access all the MIB source files from a Cis coIOS source repository
- Copies all `makefiles` and `diffs` files from the MIB repository

You can use a MIB workspace to test both Cisco IOS and non-Cisc oIOS MIBs. That is, the MIB files you are testing do not already have to exist in a C iscoIOS source repository.

To create or update a MIB workspace, follow these steps:

Step Locate a disk partition that has at least 25-30 MB of free space.

Step Decide which MIB baseline version to use.

Most MIBs depend on other MIBs; that is, they import from other MIBs. Therefore, to test a particular MIB, you must have a baseline of all the other MIBs. The baseline versions currently available are CiscoIOS Releases 11.1 and 11.2.

Note The MIB release group currently supports only Cisco IOS Releases 11.1 and later. There are no plans to support earlier releases.

If you are adding or modifying a MIB for a particular Cisco IOS version, choose that version as your baseline. If you are adding or modifying a MIB for a future release, choose the most recent version as your baseline.

If you are testing a non-Cisco IOS MIB, choose the most recent Cisc oIOS version.

Step Change into the directory in which you want the workspace to be created.

Step Run the `mibco.pl` script:

```
mibco.pl [release ]
```

release is the baseline release number. For Release 11.1, enter either 11.1 or 111. For Release 11.2, enter either 11.2 or 112. If you omit a release number, `mibco.pl` prompts you for it.

24.17 Test a MIB

After you have created a new MIB or modified an existing one, use the testing tools provided by the MIB release group to verify that the MIB contains no syntax errors. These tools include the SNMP Management Information Compiler (SMIC) MIB compiler, which is the most anal and thorough compiler known to the MIB release group. Even though your MIB gets compiled in CiscoIO S build trees, a process that commonly finds many syntax errors, many other syntax errors silently slip through this compilation phase, only to be caught by the SMIC compiler

The easiest way to test a MIB is to use the MIB release group's `update-mibs.pl` script. This script runs tests driven from a set of `makefiles`. If you understand how the `update-mibs.pl` script works, you can also choose to use `make` directly as described in the section "Use Make Directly to Generate a MIB."

To test a MIB, follow these steps:

Step Log in to a system that meets the following criteria:

- It is running SunOS 4.*x* or SunOS 5.*x* (also know as Solaris 2.*x*). Support for other operating systems will, hopefully, be added in the future

- The executable `/usr/local/bin/perl` is present. Several of the MIB release group tools are Perl scripts.
- If the Cisco IOS source code for your selected baseline version is being archived in ClearCase, ClearCase must be installed on the system.

On this system, you must have access to the following directories:

- `/nfs/csc/mib-release`, which contains the MIB release group tools and repositories.
- `/nfs/csc/smicng`, which contains the SMIC MIB compiler.

Step Add the directory `/nfs/csc/mib-release/bin` to your shell's path variable.

If you are running `csh`, use the following command:

```
set path = ( /nfs/csc/mib-release/bin $path)
```

If you are running `sh`, use the following commands:

```
PATH=/nfs/csc/mib-release/bin:$PATH
export PATH
```

Step If you have not already created a MIB workspace, as described in the section "Create or Update a MIB Workspace," do so now.

Step Copy your new or modified MIB (the `*.my` files) into the `test-mibs` directory in your workspace.

Step Change your current directory to the root of your workspace.

Step If you are testing changes to an SNMPv1 MIB, issue the following command:

```
update-mibs.pl -1
```

Step If you are testing changes to an SNMPv2 MIB, issue the following command:

```
update-mibs.pl
```

24.18 Analyze Test Results

When the `update-mibs.pl` script completes, it informs you that either all the MIBs successfully passed the tests or there was a failure. If there was a failure, the script directs you to the log file for details of the failure. After checking the log file, edit your MIB files in the `test-mibs` directory and run the script again.

All MIB files in the `MIBS` and `test-mibs` directories are passed through a publication filter before they are run through SMIC. For example, your `test-mibs/TEST-MIB.my` is passed through a filter to create `v2/TEST-MIB.m` , which is the file that is run through SMIC. The filtering process probably removes some lines from your MIB file. The SMIC error and warning messages that are displayed contain line numbers that correspond to the lines in the `v2/TEST-MIB.my` file, not to the lines in your `test-mibs/TEST-MIB.my` file. Therefore, you will need to consult the files in the `v2` directory (or the `v1` directory if you are testing an SNMPv1 MIB) to determine the line on which SMIC is reporting an error. However, we recommend that, when correcting errors, you edit only the files in the `test-mibs` or `MIBS` directory.

24.19 Determine Whether You Have an SNMPv1 or SNMPv2 MIB

The section provides some tips for determining whether you have an SNMPv1 MIB or an SNMPv2 MIB.

Several new macros have been defined for SNMPv2. You have an SNMPv2 MIB if you can find any of the following strings in your MIB:

- `MODULE-IDENTITY`
- `MODULE-COMPLIANCE`
- `OBJECT-GROUP`
- `NOTIFICATION-TYPE`
- `TEXTUAL-CONVENTION`

Also, MIB objects defined in an SNMPv1 MIB should have an `ACCESS` clause, while MIB objects defined in an SNMPv2 MIB should have a `MAX-ACCESS` clause.

If you still cannot determine which type of MIB you have, contact the MIB release group at the e-mail alias `mib-release`.

24.20 Generate an SNMPv1 Version of an SNMPv2 MIB

Not all customers can use SNMPv2 MIBs. For these customers, Cisco provides SNMPv1 versions of the MIBs. If your MIB is part of a shipping CiscoIO S release, the MIB release group creates the SNMPv1 versions and provides them to the customers. If your MIB is a non-Cisco I OS MIB or if circumstances require you to provide a customer-special release, you must create the SNMPv1 version of the MIB.

You can create an SNMPv1 MIB in one of the following ways:

- Run the `update-mibs.pl` script as described in the section "Test a MIB," following the procedure for testing an SNMPv1 MIB.
- Use `make` directly as described in the section "Use Make Directly to Generate a MIB."

24.21 Use Make Directly to Generate a MIB

You can run `make` directly to generate a MIB instead of using the `update-mibs.pl` script. If you choose to do this, you should first understand the following:

- How `mibco.pl` works; see `http://wwwin-eng.cisco.com/Eng/IOS/SNMP_WWW/Mib-Release/under-covers.html`
- What the `make` targets are; see `http://wwwin-eng.cisco.com/Eng/IOS/SNMP_WWW/Mib-Release/make/make-targets.html`

24.21.1 Use Make Directly to Generate an SNMPv2 MIB

To use the `make` command directly to generate an SNMPv2 MIB, follow these steps:

Step Change into the root directory in your MIB workspace.

Step Change into the `v2` directory:

```
cd v2
```

Step Test the SNMPv2 MIB to ensure that its syntax is correct:

```
make depend
make all
```

24.21.2 Use Make Directly to Generate an SNMPv1 MIB

To use the `make` command directly to generate an SNMPv1 MIB from an SNMPv2 MIB, follow these steps:

Step Change into the root directory in your MIB workspace.

Step Convert the SNMPv2 MIBs into SNMPv1 MIBs:

```
make depend
```

Step Test the SNMPv2 MIBs to ensure that its syntax is correct:

```
cd v1
make all
```

The resulting converted SNMPv1 MIB is placed in `v1/`*`mib-name`*`-V1SMI.m` .

Optionally, you can generate a single SNMPv1 MIB rather than generating them all. Before doing this, you should first familiarize yourself with the `make` targets.

24.21.2.1 Example: Use Make to Generate an SNMPv1 MIB

The following example shows how to generate an SNMPv1 version of the SMNPv2 MIB `CISCO-FLASH-MIB.my`:

```
make depend
cd v1
make CISCO-FLASH-MIB-V1SMI.v1
```

or

```
make CISCO-FLASH-MIB-V1SMI.smicng
```

The resulting converted SNMPv1 MIB is placed in the directory `v1/smicng` and is named `CISCO-FLASH-MIB-V1SMI.m` .

24.22 Publish a MIB

Publishing a MIB means making it accessible to customers. Cisco provides MIBs to customers through an FTP server and Cisco Connection Online (CCO). MIB administrators are responsible for releasing MIBs through FTP. The files in the FTP server are distributed using `rdist` to CCO on a nightly basis. This process is handled by the people at the e-mail alias `cco-team`.

MIBs for the latest shipping release are archived in the directory `ftp://ftpeng.cisco.com/pub/mibs`. The contents of this archive are mirrored in CCO.

MIBs for a beta release are archived in the directory `ftp://ftpeng.cisco.com/beta`*`xxx`*`_dir/mibs_`*`xxx`*, where *xxx* is the major release number without the periods. For example, the MIBs for the beta version of Cisco IOS Release 11.2 are archived in `beta112_dir/mibs_112`.

24.22.1 Prerequisites for Publishing a MIB

To get a new MIB published, the following prerequisites must be satisfied:

- The MIB must be supported in a release that is scheduled to be shipped to customers.
- The MIB must be committed to the source repository appropriate for that release.

- The MIB must cleanly pass the tests in the MIB release group's test suite.
- You must send a request to the e-mail alias `mib-release`. The request should clearly state that you are requesting MIB publication and should include the following:
  - MIB filename.
  - Release in which the MIB is supported.
  - Short description of the functionality provided by the MIB. For examples of short descriptions, see the file `ftp://ftpeng.cisco.com/pub/mibs/v2/readme`.

To have your MIB published in a timely fashion, notify the MIB release group *before* the day that beta starts shipping or *before* a release's first customer ship (FCS) release.

Once the MIB release group has published a MIB for any release, you do not need to request publication for later releases. The MIB release group assumes that the MIB should continue to be published for all following releases. However, if this assumption is incorrect, notify the MIB release group.

24.23 MIB-Related Files

24.23.1 File Locations

All files and tools supported by the MIB release group—with the exception of the SMIC toolset—are located in the directory `/nfs/csc/mib-release`. This directory contains the following subdirectories:

- `bin`—Contains the MIB release group's toolset
- `lib`—Contains Perl libraries used by the toolset
- `docs`—Contains documentation about the toolset

The SMIC toolset, which contains the SMIC MIB compiler and other related files is located in the directory `/nfs/csc/smicng`.

24.23.2 MIB Repository and Workspace

A *workspace* is your personal copy of the MIB repository. As with the Concurrent Versions System (CVS) or ClearCase, you have a private directory where you can access files from the repository. You then perform your builds, compiles, tests, and other operations from within that workspace. The repositories are located in `/nfs/csc/mib-release/`*xxx*, where *xxx* is the major release number (for example, 112 for Cisco IOS Release 11.2).

Both the structure and contents of a MIB repository and a MIB workspace are identical. When you create a MIB workspace, the relevant files, including the `makefiles` and `diffs`, are copied from the appropriate repository into your private workspace. Every operation that you can perform in a workspace can also be performed in a repository.

Only MIB release group administrators can update the files in the MIB repository.

24.23.3 Files in the MIB Repository and Workspace

The MIB repository and workspace contain the following types of files:

- MIB files, which are the actual MIBs themselves.

- `makefiles`, which are used to drive much of the MIB release group's toolset.
- `diffs` files. Many times MIBs or generated files need to be tweaked to deal with quirks in the MIB compiler. The `diffs` files record the tweaks that are necessary. Your MIB files are patched from these `diffs` files before being run through the MIB compilers. Under normal circumstances, you should never have to create or modify any `diffs` files.
- `schema` files. All Cisco MIBs are converted to SunNet Manager's `schema` file format. Under normal circumstances, you should never have to generate `schema` files.
- SMIC support files, which are auxiliary files that must be created before a MIB file can be compiled with SMIC. The creation of these files is handled entirely by the `makefiles`. If you are interested in how SMIC works, look at the SMIC support files.

24.23.4 Directory Layout for MIB Repository and Workspace

A MIB repository or workspace has the following directory structure. Table 24-3 explains the contents of the directories.

```
test-mibs/
MIBS/
v2/
  diffs/
  smicng/
    diffs/
v2-to-v1/
  diffs/
  smicng/
    diffs/
v1/
  diffs/
  smicng/
    diffs/
schema/
  diffs/
  schema/
  oid/
  traps/
```

Table 24-3 MIB Repository and Workspace Directory Layout

Directory	Contents
`test-mibs/`	MIBs you are testing.
`MIBS/`	Full set of MIBs that have been committed to the source repository. Usually MIBs depend upon other MIB files; that is, they `IMPORT` from them. Therefore, you must have a baseline of all the MIB files in order to test any particular MIB.
`v2/`	SNMPv2 MIBs to be compiled. This directory is populated from files in the `MIBS` and `test-mibs` directories. Compilation takes place in the `v2/smicng` directory.
`v2-to-v1/`	SNMPv2 MIBs to be converted to SNMPv1 MIBs. This directory is populated from files in the `v2` directory. The actual conversion takes place in the `v2-to-v1/smicng` directory. The `v1` version of *xxx*`-MIB.my` is in the file *xxx*`-MIB-V1SMI.m` .
`v1/`	SNMPv1-style MIBs to be compiled. This directory is populated from files in the `MIBS`, `test-mibs`, and `v2-to-v1/smicng` directories. (SNMPv1 MIBs converted from SNMPv2 MIBs are taken from the `v2-to-v1/smicng` directory.) Compilation takes place in the `v1/smicng` directory.

Directory	Contents
`schema/`	MIB files that need to be converted to SunNet Manager `schema` files. Converting the file *`xxx`*`-MIB` gets generates three files: `xxx-MIB.schema`, *`xxx`*`-MIB.oid`, and *`xxx`*`-MIB.traps` (if any SNMP traps are defined in the MIB), which are located in the subdirectories `schema`, `oid`, and `traps`, respectively.
`.../diffs/`	Patches to be made to files in the parent directory. For example, `v2/diffs` contains patches to be made to the SNMPv2 MIBs. `diffs` subdirectories exist in the `v1`, `v2`, `v2-to-v1`, `schema`, and all `smicng` directories.
`.../smicng/`	SMIC support files. You run the SMIC compiler from this directory. `smicng` subdirectories exist in the `v1`, `v2`, and `v2-to-v1` directories.

Most directories in a MIB repository or workspace contain a `makefile`, which contains the rules common to the directory in which it is contained. Table 24-4 explains the files related to the `makefiles`.

Table 24-4 makefile Structure

File/Directory	Contents
`makefile.defs`	Common definitions used by all `makefiles`, for example, program names, locations, and options, and lists of MIB files to process or ignore. This file is included by all `makefiles` in the tree.
`makefile.common`	Rules common to all `makefiles` in the tree. This file is included by all the `makefiles`.
`makefile.mib`	Rules common to the `v2`, `v2-to-v1`, and `v1` directories. For example, the rules for copying MIB files from the `MIBS` and `test-mibs` directories and passing them through the `publish-mib` filter are defined here. This file is included by all the `makefiles` in these directories.
`makefile.smic`	Rules common to the `*/smicng` directories. For example, the rules for running the SMIC tools are defined here. This file is included by all the `*/smicng/makefiles`.

IF-MIB

If you are adding a new interface or subinterface type (for IF-MIB), or a new card/port/chassis type (for Entity-MIB), there are a series of integration requirements for SNMP that you must address.

25.1 Supporting Subinterfaces in IF-MIB

With the addition of the IF-MIB (RFC 2233), the Cisco IOS software can now support subinterfaces in the interfaces group of MIB-II. To provide support for these new subinterfaces, you need to understand four key IF-MIB tables, the subiabtype data structure, and the functions in the IF-MIB API. First, this chapter describes these components, then tells you how to use them in registering or deregistering sublayers. Then a sample implementation is provided, which uses Frame Relay sublayers. At the end of the chapter, there is some information about link up/down trap support.

25.1.1 Tables

All newly registered interfaces and sublayers should support four tables of the IF-MIB:

- `ifTable`
- `ifXTable`
- `ifStackTable`
- `ifRcvAddressTable`

Read RFC 2233 to understand how this support should be carried out and which constraints are set by the IF-MIB and which by the sublayer media type.

Note Cisco subinterfaces do not directly correlate to IF-MIB sublayers.

25.1.2 API

A series of files holds the support for registering, updating, and deregistering subinterfaces in the `IF-MIB: snmp/ifmib_registry.reg, snmp/ifmibapi.[ch], h/snmp_interface.h,` and the `ifType` file.

25.1.2.1 snmp/ifmib_registry.reg

This file holds the external registry functions for all IF-MIB-related calls into the IF-MIB API. Use the functions in this registry. Only the `files h/snmp_interface.h` and `snmp/ifmib_registry.h` should be included in any files requiring support for this API.

25.1.2.2 snmp/ifmibapi.[ch]

These files contain the internal support for the IF-MIB API. Do not use these functions directly; use the service points provided above. Reading these files gives you a fuller understanding of the IF-MIB API.

25.1.2.3 h/snmp_interface.h

This file contains the subiabtype data structure. Be sure that you study and understand this structure. It is the one that you use to pass information across the IF-MIB API.

25.1.2.4 ifType

The `ifType` for a given subinterface is now supported in the `IANAifType` Textual Convention (TC). See `MIBS/IANAifType-TC.m` . If an appropriate value for `ifType` for your sublayer type is not present in this TC, you should request a new value from the Internet Assigned Numbers Authority (IANA). This request should not be made lightly as it requires you to design the IF-MIB requirements in each table for your new media type. Wherever possible, use the Internet Engineering Task Force (IETF) media MIB guidelines. Also, verify that the `IANAifType-MIB` is current by checking the IETF Web site.

25.2 Adding Support to Register or Deregister Sublayers

The primary data structure used for sublayers is the `subiabtype`, as defined in `h/snmp_interface.h`. You should use this structure for any interfaces or sublayers that are not `hwidb`-based. The rest of this section describes how to correctly support your new sublayer in the IF-MIB.

25.2.1 Adding to Service Points

As can be seen in `snmp/ifmib_registry.reg`, there is a series of service points to which a new sublayer type must add support. These service points provide unique functions that the IF-MIB can call to update its information about a given sublayer. All `RETVAL` or `LIST` type service points use the `ifType` as the selection criteria. Note that most of these functions have a `TEST` phase (associated with the SNMP test phase) and a `SET` phase. The current service points that may need to be supported are these:

- `reg_add_ifmib_get_operstatus()`

  This service point will retrieve the `ifOperStatus` for the sublayer. There is a default function, `ifmib_get_operstatus_default()`, which may be used if the new sublayer `ifOperStatus` is reflected in the `idb->subif_state` value. If this is not true, then the new sublayer must add its own function.

- `reg_add_ifmib_get_adminstatus()`

This service point will retrieve the `ifAdminStatus` for the sublayer. There is a default function, `ifmib_get_adminstatus_default()`, which may be used if the new sublayer `ifAdminStatus` is reflected in the `idb->subif_state` value. If this is not true, then the new sublayer must add its own function.

- `reg_add_ifmib_admin_change()`

  This service point will allow the `ifAdminStatus` to be writable. If it is appropriate to administratively put the sublayer up or down, this service point must be supported. There is a default function, `ifmib_admin_change_default()`, which may be used if the new sublayer can use the `shutdown_subif()` function to control the interface state. If this is not true, then the new sublayer must add its own function if `ifAdminStatus` is to be fully supported as read-writable.

- `reg_add_ifmib_cntr32()`

  This service point will retrieve the requested `32-bit` counter value for the sublayer. The counter types are for each counter in the `ifTable` and `ifXTable`. There is one counter function for all counters, with an appropriate `countertype` passed in as the parameter to select which counter value to return. The counter types are specified in `h/snmp_interface.h` as `ifmib_cntr_t enum`. There is no default function for this service point. If any counters in the `ifTable` or `ifXTable` are supported for this sublayer, then one appropriate function must be added to support these counters.

- `reg_add_ifmib_cntr64()`

  This service point is the `64-bit` equivalent `Hispeed` counter function for the sublayer. Check with RFC 2233 to see if the sublayer should support the `HCCounters` for the `ifXTable`. If these `64-bit` counters must be supported, then the appropriate functions must be added to support these HC counters. There is no default function.

- `reg_add_ifmib_rcvaddr_screen()`

  This service point screens additions and deletions to the `ifRcvAddressTable`. It is only appropriate if the `ifRcvAddressTable` entries for this sublayer type are writable via SNMP.If the entries are not writable via SNMP, but additions and deletions are made via another source (i.e. the CLI, or as a result of changes to a media-specific MIB) please use `reg_invoke_create_rcvaddr()` or `reg_invoke_delete_rcvaddr()` to make changes to this table.

- `reg_add_ifmib_stack_screen()`

  This service point screens additions and deletions to the `ifStackTable`. It is only appropriate if the `ifStackTable` entries for this sublayer type are writable via SNMP. If the entries are not writable via SNMP but additions and deletions are made via another source (that is, the CLI or as a result of changes to a media-specific MIB), please use `reg_invoke_create_stacklink()` or `reg_invoke_delete_stacklink()` to make changes to this table.

- `reg_add_ifmib_add_subif()`

  This service point registers a sublayer. The selection is based on `hwidb->enctype`. It can only be used for sublayers that have a unique `enctype`; otherwise, use `ifmib_register_subif()`.

- `reg_add_ifmib_update_ifAlias()`

  This service point is used to manipulate `ifAlias` value for `subiabs`. It switches based on `ifType`. If the underlying sublayer structure is a `swidb`, then the default function will handle the `ifAlias` update via `swidb->description`.

- `reg_add_fmib_get_last_change()`

This service point retrieves the `sysuptime` for the last state change on a sublayer. It will switch based on `ifType`. It is up to the media code developer to keep the last change `timestamp` updated whenever the interface changes state (`ifOperStatus` changes). There will be a default function available for sublayers based on `swidbs`, which will pick up the `timestamp` kept in the `swidb` structure.

- `reg_add_ifmib_get_if_speed()`

  This service point retrieves the `ifSpeed`. Note that `MAXULONG` must be returned if the `ifSpeed` is greater than `MAXULONG`. If the underlying sublayer structure is `swidb`, then the default function will return `ifSpeed` based on `swidb->visible_bandwidth`.

- `reg_add_ifmib_get_if_highspeed()`

  This service point retrieves the `ifHighSpeed`. If the underlying sublayer structure is `swidb`, then the default function will return `ifHighSpeed` based on `swidb->bandwidth`. Note that `ifHighSpeed` is defined in Mbps and, as such, is accurate to +/- 500,000 bits per second.

25.2.2 Registering a Sublayer

Once a sublayer is created, you should register it with the IF-MIB. Registration is accomplished with the `reg_invoke_ifmib_register_subif()` function.You must fill in the following parts of the `subiab` appropriately before calling the `reg_invoke_ifmib_register_subif()` function:

```
subiab->state
subiab->if_descrstring
subiab->if_name
subiab->ifPhysAddr
subiab->ifPhysAddrLen
subiab->ifType
subiab->maxmtu
subiab->idb_type
subiab->connector_present (11.3P)
subiab->link_trap_enable (11.3P)
```

Note Even if `ifPhysAddr` is inappropriate for the new sublayer type, these values should be filled in as zero.

In some cases, the sublayer may be created, but the `ifPhysAddr` is not yet known. In this case, the sublayer can be registered with zero `ifPhysAdd` , but it is up to you, the developer, to determine when the address is known and to make a call to `reg_invoke_create_rcvaddr()` to add this new address to the `ifRcvAddressTable`, as well as to retrieve the appropriate `snmpidb` structure and fill in the `subiab->ifPhysAddr` and `subiab->ifPhysAddrLen`.

25.2.3 Deregistering a Sublayer

Once the sublayer has been deleted, it should be deregistered from the IF-MIB with `reg_invoke_ifmib_deregister_subif()`. This function will remove the `ifTable`, `ifXTable`, `ifRcvAddressTable`, and `ifStackTable` entries associated with the sublayer. The associated `snmpidb` and `subiab` memory will be free'd.

25.2.4 Modifying the ifRcvAddressTable

This table can be modified directly via the IF-MIB (writing to the table), or indirectly via the CLI or a media-specific MIB. If the sublayer addresses can be modified via the IF-MIB, `reg_invoke_rcvaddr_screen()` must have an appropriate function to verify that this modification can take place. Remember that this function can add or delete an entry. It also has a test mode, wherein the changes requested are verified without actually modifying the table. This is required for the test phase of an SNMP MIB write (`k_ifRcvAddressEntry_test()`), and must be supported in the sublayer screen function. If the modification to the table is made outside the IF-MIB, it is assumed that the screening process has already taken place, and a call can be made directly to `reg_invoke_create_rcvaddr()` or `reg_invoke_destry_rcvaddr()`). These functions simply add or remove entries into the `ifRcvAddressTable` with little sanity checking.

25.2.5 Modifying the ifStackTable

This table can be modified directly via the IF-MIB (writing to the table), or indirectly via the CLI or a media-specific MIB. If the sublayer stack links can be modified via the IF-MIB, the `reg_invoke_stack_screen()` must have an appropriate function to verify that this modification can take place. Remember that this function can add or delete an entry. It also has a test mode, wherein the changes requested are verified without actually modifying the table. This is required for the test phase of an SNMP MIB write (`k_ifStackEntry_test()`) and must be supported in the sublayer screen function. If the modification to the table is made outside the IF-MIB, it is assumed that the screening process has already taken place, and a call can be made directly to `reg_invoke_create_stacklink()` or `reg_invoke_destroy_stacklink()`. These functions simply add or remove entries into the `ifStackTable` with little sanity checking.

25.2.6 Sparse Table Support

The IF-MIB contains many objects that might not be appropriate for a given sublayer type. As such, there is inherent support in the IF-MIB method routines to support a sparse table implementation. There is also support on the CLI to turn off this sparse table support, effectively returning zero/Nullstring where needed to provide full tables.

25.3 Sample Implementation: Frame Relay Sublayers

A sample implementation is available using Frame Relay sublayers. This is a simple example as there are no physical address equivalents for Frame Relay, and the `ifStackTable` is read-only for these entries.

25.3.1 Adding Service Points: Frame Relay

A `counter32` service point was added for the Frame Relay counters:

```
reg_add_ifmib_cntr32(D_ifType_frame_relay,fr_subif_cntr32fn, "fr_subif_cntr32fn") ;
```

This was added in `wan/sr_frmib.c - init_frmib()`. Note that the function is added with the `ifType` of `frame_relay` as the selection criteria.

Frame Relay uses the defaults for getting `ifOperStatus`, `ifAdminStatus`, and changing `ifAdminStatus`, so no specific code is added for these.

Taking a look at the counter function `fr_subif_cntr32fn()`, one can see there is minimal support for counters in Frame Relay currently: only the `ifInOctets` and `ifOutOctets` are present. All other queries will return `IF_CNTR_NOT_AVAIL` as an error response. The counter value itself is passed to this function as a pointer and filled in with the current counter value.

25.3.2 Registering a Sublayer: Frame Relay

Frame Relay sublayers are registered as they are created. This can be seen in `if/network.c`. The function `reg_invoke_ifmib_add_subif()` is used. Looking at the function that is actually called here— `snmp/sr_ifmib.c - ifmib_add_subif()`—one can see that the appropriate values are filled into the `subiab` data structure and passed to the `reg_invoke_ifmib_register_subif()`. This function will take care of allocating memory for the `snmpidb` and the `subiab` pointed to by this function, so the `subiab` structure in `ifmib_add_subif()` can be a local data structure.

Also notice that this function adds the stacklink to the `ifStackTable` via `reg_invoke_ifmib_create_stacklink()`. In this simple case, it is known that the sublayer sits directly on the `hwidb` interface. In the more generic case, the sublayer may sit on another sublayer, or perhaps a many-to-one relationship exists. The developer must make the appropriate calls to the stack functions to add these links.

25.3.3 Deregistering a Sublayer: Frame Relay

This is a simple task of calling `reg_invoke_ifmib_deregister_subif()` and passing it the correct `ifIndex` for the interface to be deregistered. Note that `subiab` and `snmpidb` will be free'd. `NULL` checks are your friend.

Note There may not be one single place where a sublayer is created or destroyed. It is up to you to scope out all possible places for the sublayer to be modified and make appropriate calls to the IF-MIB service points.

25.4 Link Up/Down Trap Support

Support for link up/down traps per sublayers has been added to 11.3P,but it remains that most sublayer types should default to link traps disabled (that is, never generated). Any deviation should be reviewed with the SNMP group before being implemented. Contact the group on the `snmpv2-dev` email list.

PART 7

Other Useful Information

Scalable Process Implementation

This chapter was originally Cisco IOS Technical Note #5, written in January 1997 by Dave Katz, an engineer on the IOS Protocols team. It was incorporated into this guide in September 1998.

The Cisco IOS kernel has good potential for supporting scalable, well-behaved processes that can support very large networks. This chapter addresses shortcomings in the code that interfere with developing scalable processes and describes ways to avoid these shortcomings.

26.1 Introduction

The Cisco IOS kernel has a lot of potential for supporting scalable, well-behaved processes that can support very large networks. Unfortunately, our track record in producing such software has been spotty. In the five years that I have worked on Cisco IOS code, I have seen (and fixed) lots of code that had common mistakes. The intent of this document is to discuss these shortcomings and describe ways to avoid them, in order to improve the scalability of the product without requiring massive rewrites under pressure.

Because the bulk of my experience is in the area of routing protocols, I will be using them as examples. They are also quite illustrative in that they can be quite CPU and bandwidth intensive, not surprisingly the two biggest problem areas in writing good Cisco IOS code. This comment should be interpreted neither as an indictment of our routing protocol implementations, nor as an acceptance of the status quo in other parts of the code. The lessons to be learned from routing protocols apply across the product line.

26.2 The Typical Scenario

Routing protocols are generally pretty complex beasts. As such, the effort required simply to understand a protocol well enough to implement it ends up burning the majority of the brain cells of the original implementer.

Because a protocol itself is complex, there is a natural desire to implement it in the most straightforward way possible. This is a desirable engineering practice, because the initial goal is correctness rather than efficiency. Furthermore, premature optimization is the cause of more programming sins than almost any other primal urge. It is much more sensible to figure out where the hot spots are after you gain some experience with the implementation. Only then should you rework or reimplement as necessary.

In practice, however, we have tended to take this mindset to its unfortunate extreme. The initial implementation tends to be *so* straightforward that architecturally it does not lend itself to later improvement.

The initial implementation is written, tested, and then shipped even though it is in fact a prototype. The developer makes many hollow promises about "fixing it later," but the pressure of deadlines, management, and the next project—and in the case of too much success, the pressure of hot sites from the field—makes it infeasible to return to the scene of the crime. The prototype ships, seems to work, and everyone is happy

The customers use the new code, and like it, and use it some more. They build bigger and bigger networks. Pretty soon, the nonlinearities in the implementation start to manifest themselves. This occurs most often in the form of high CPU utilization and then CPUHOG indications, or huge amounts of control traffic, or both. Sometimes, the code is metastable, moving quickly from being well-behaved to having sudden spasms. If left untreated, the code moves on to progressive widespread network instability and collapse. This is *not* fun.

A series of quick patches is then applied. These patches treat the symptoms, but most often do not treat the root cause, and might in fact create more serious and tricky problems because the process was never designed to get big. The code and the coder are tied in knots. The developer desperately wants to work on something else—and sometimes succeeds, much to the chagrin of the unfortunate soul who inherits the code.

Meanwhile, we give the customers some absurdly conservative numbers for the maximum network size, number of neighbors, and so forth, in order to keep their networks out of the failure regime. The competition chuckles at these numbers (but often has similar problems because they are not usually much smarter than we are; their networks are just smaller), and the customers are unimpressed.

We then undertake either a slow, drawn-out, painful process, or else a fast, hurried, painful process, to rewrite what needs to be rewritten. If things get bad enough, we commit 40,000-line patches into maintenance releases. Been there, done that.

This kind of insanity is avoidable if we can strike the right balance between simplicity and extensibility. The essential ingredient is to understand what the process is going to look like when things get complicated, to structure it accordingly while it is still simple, and to rewrite it if (and when) it turns out to be wrong anyhow.

26.2.1 Specific Problems

There are a number of specific problems seen in naively implemented processes. They are often interrelated and might stem from similar faults. Some symptoms are more universal and appear as side effects of a multitude of problems. These problems include the following:

- CPU Utilization
- Excessive Protocol Traffic
- Adjacency Failures
- Brittle Networks
- Random Squirrely Failures
- Pathological Process Interaction

26.2.1.1 CPU Utilization

An early indication of implementation problems is CPU utilization difficulty. The CPU might peg at 100% for significant periods of time, and CPUHOG errors might also result. Note that high CPU utilization in itself is not necessarily bad or even disruptive if the processor is being appropriately shared, but it should invite further investigation. CPUHOG errors are 100% evil, however, and indicate serious flaws in overall code architecture and poor choices in data structure and algorithms.

26.2.1.2 Excessive Protocol Traffic

Many protocols are simply chatty, and we are stuck with them. However, most modern protocols are not inherently chatty, and most can be equipped with sufficient nerd knobs to allow a trade-off between chattiness and convergence rate. When links light up with 100% utilization because of control traffic, this situation is almost always avoidable, even if it is not incorrect. All that control traffic displaces data traffic, and the network just is not healthy. A related problem is the excessive loss of control traffic, which can trigger retransmissions in some protocols. The traffic loss is usually an indicator of poor implementation, either in the sender or receiver. Such loss can seriously impact network convergence, and thus overall stability, and often incurs significant CPU load as well.

26.2.1.3 Adjacency Failures

An extremely serious symptom is the loss of neighbor connectivity in protocols that have adjacency maintenance functions. When neighbors drop and come back, a big impulse is usually thrust into the network. If stability is critical enough, the flood of control traffic itself can cause further neighbor loss. Then what you have is a network that has fallen and cannot get up. This usually results in VPs calling other VPs, and your being awa kened at 4a.m. Although fascinating in the same morbid sense as a multicar pileup, this scenario is one to be avoided.

26.2.1.4 Brittle Networks

Brittle networks are a little more difficult to describe. Basically, this is a situation in which the network does not degrade gracefully. It might recover just fine, but it tends to be either very quiet and stable, or very tempestuous. This brittleness usually indicates poor control schemes in the system. For instance, if a system becomes more efficient as it becomes loaded, it has a chance of recovering smoothly. If it becomes *less* efficient under load, the load will increase even faster and things get unpleasant.

26.2.1.5 Random Squirrely Failures

Random squirrely failures often happen when hapless attempts are made at addressing some of the other symptoms. Symptoms of these failures include buffer and memory management problems, `NULL` dereferences, and pointers overwritten by the “poison” pattern (`0D0D0D0D`). These problems usually result from either poor organization of resource management or race conditions introduced when trying to fix CPU utilization problems. They can be tough to diagnose and nearly impossible to reproduce.

26.2.1.6 Pathological Process Interaction

Because the Cisco IOS environment is one of shared resources (memory, CPU, buffers, and bandwidth), one misbehaving process can trigger failures in another, often in ways that are not at all obvious.

26.3 Addressing the Problems

There is only one way to avoid this litany of mistakes—by building software that is structured cleanly and robustly. This goal must be addressed from the very beginning of a project, in the way that the process (or often multiple processes) are designed, how the functionality is divided up among processes, how data is organized, and so forth. The beginning of a project is the time to ask

questions like, “What would happen if there were 1000 interfaces?”, “What if the system had 500 neighbors?”, and “What if the system cannot keep up with incoming control traffic or with the rate at which control traffic is being generated?”

In this section I examine a number of strategies for avoiding the kinds of symptoms described in the previous section, and I attempt to provide some insight into the kinds of problems that result when things are not done carefully.

26.3.1 Process Structure

The first question is a big one—how many Cisco IOS processes are actually needed to perform the task? When the size of the network is small enough, any problem can be addressed with a single process. However, Cisco IOS has some features (or a lack thereof) that almost always cause difficulties if only a single process is used.

At this time, the Cisco IOS scheduler does not provide any preemption mechanism. This means that each process is responsible for releasing the CPU on a regular basis and for doing its own internal scheduling for handling events.

Most protocols have two kinds of subtasks, those that are CPU intensive and those that are time critical. For instance, a link-state routing protocol might require several CPU seconds to calculate routes over a large routing database, and it might also require that hello packets be exchanged within a particular time period to maintain neighbor adjacencies. It does not take much thought to realize that if both of these operations are done in the same process, the implementation either will be hideously complex (requiring some kind of internal preemption) or will simply fail miserably when the CPU-heavy portion takes so long that the time-critical portions do not happen quickly enough.

These requirements immediately lead us to the idea of using multiple processes and having the scheduler take care of the scheduling. (That’s its job, of course.) Preemption of the CPU-intensive portion must still be done explicitly, but the time-critical portion will fend for itself so long as nobody hogs the CPU. This is necessary but not sufficient.

This process design more explicitly raises an issue that was already lurking in the background, but that few have noticed—atomicity. Certain operations are implicitly assumed to be atomic—they are executed to completion with the guarantee that all data used for the operation are unchanged. Look back at our expensive link-state calculation, for example. To be a good Cisco IOS citizen, the code performing this calculation must relinquish the processor regularly, on the order of every 100 m illiseconds or so. However, the route calculation code almost certainly assumes that the link state database does not change while the calculation is taking place. When the first `CPUHOG` happens because the route calculation is taking too long, someone will start putting suspends in the code to fix it. But this opens up the distinct possibility that another process will jump in and modify the data structures on which the route calculation is relying.

Things get even more complicated when there are other paths into the code from other process threads. Nearly every area of code in the system has a path into it from the EXEC process thread, which is used when someone is configuring the system. The configuration might change whenever a process suspends! Also common are callbacks from other processes. For example, when routes are redistributed between protocols, the code in the receiving protocol is often executed on the thread of the sending protocol. The old and dreaded “active timer” system adds yet another process thread from which unintentional consequences might result.

There are two key concepts that come to bear in this situation: atomicity and serialization. They sometimes go together, and sometimes are at odds, and the art of process design comes in finding the appropriate balance.

The most obvious way to make an operation atomic is simply to refuse to give up the CPU during the operation. Assuming that we are not worried about interrupt code changing things, this method means we simply do not suspend the process until we are through. This is easy, is often done, and causes CPUHOG errors. Even if the code path looks small enough at the top, it may be calling procedures that turn out to be quite expensive.

So how do we keep things atomic but not cause CPUHOGS? An operation is guaranteed to be atomic, even if the process suspends, so long as there is *no* path from *any* other process that can change things. (We have seen that this requires the utmost care.) This guarantee can be achieved by serializing any changes so that they do not actually take place until the atomic operation has completed.

Serialization of Cisco IOS processes can be done in a couple of ways. The simplest way is to put all the operations that could change the critical data structure into the same process as the atomic operation. The process main loop has its own little scheduler. As long as the atomic operation is running—even if it is suspending—control does not return to the process main loop until the atomic operation is complete.

Typically, there are events that take place in another process that affect the critical data structure. For example, the time-critical process might detect a new neighbor or the failure of an existing neighbor. The neighbor change necessitates a change in the data structure over which our atomic operation is being done. In this case, serialization can be accomplished by having the time-critical process post an event onto an event queue in the second process. The second process then processes the event, making the necessary changes, after it completes its atomic operation.

This latter scheme effectively defers the handling of an event until some time in the indeterminate future. This is all well and good for our atomic operation, but it might break an assumption of atomicity that was part of the event. For instance, take a look at our configuration example. There is some assumption that the processing of a configuration command takes place immediately, before the next command prompt is presented to the user. By enqueuing an event and then returning, we let the user enter the next command before the previous one might have been processed. If some error condition is detected when the event is finally processed, we can only complain after the fact. Furthermore, if some configuration changes are deferred and some are done immediately, we might accidentally change the order of operations. There is no easy answer to this one, other than being careful to think through the consequences.

26.3.2 Stability through Rate Control

One of the Holy Grails of protocol design and implementation is fast convergence. Indeed, this is the stuff from which marketing campaigns are built. Sometimes, there is customer pressure to make things go faster, for no reason other than that it must be better (as opposed to fixing any actual operational problem). Experience has shown, however, that being as fast as possible is not necessarily a good thing.

Networks are distributed, loosely coupled systems that exhibit large-scale behavior that is a product of the behavior of the individual systems in that network. However, there is a lack of feedback in the network. A single machine cannot tell, particularly on an instantaneous basis, how the network as a whole is behaving. Furthermore, any attempts to signal this information becomes a part of the control stream and changes the behavior (the old Heisenberg uncertainty principle).

Because there is little feedback, a single system must be careful how it impacts the rest of the network (which it does by sending control traffic). It is easy to see that if a system generates control traffic at a rate faster than the network can absorb it, bad things will happen. Trying to be “as fast as possible” translates into “send control traffic as fast as possible,” which is at cross-purposes with stability.

For the network to be stable, all traffic generation must be controlled and controllable. Packets must be transmitted at a rate that is in line both with the available link bandwidth and with the reception bandwidth of the guy at the other end of the link. This requirement must be an integral part of the implementation—for instance, done using a per-interface managed timer that fires at regular intervals to trigger transmission. If this infrastructural work is done in the implementation, it later becomes possible to vary the transmission rate automatically and manually to optimize the network.

In addition to providing control over packet transmission rates, it is often useful to provide knobs to control the rate of CPU-intensive operations, if this is feasible. For instance, in link-state protocols, the interval between successive route calculations can be controlled. Any topological changes that occur in between these calculations are noted in the link-state database, but otherwise incur very little CPU penalty.

Rate control comes at a price, of course, which is the rate of network convergence. However, extremely rapid convergence is overrated. If the network converges quickly enough so that the user does not have a chance to call the network administrator, that is quick enough. It is also the case that as networks get larger, they *will* converge more slowly. Fact of life. However, a stable but slightly pokey network is vastly preferable to a lightning-fast network that melts down periodically

26.3.3 Avoiding Receive Buffer Starvation

The Cisco IOS kernel uses a credit scheme for allocating I/O buffers to interfaces. When a packet is received on an interface, the interface loses one credit. If all credits are consumed, the interface drops incoming traffic until the credits are returned.

Typically, control protocol packets are processed to completion and then returned, at which point the credit is returned. This means that the input credits are reduced while packets are waiting to be processed.

For complex control protocols, the time required to process an incoming packet might be arbitrarily long. For example, IS-IS link-state packets (LSPs) must be queued while the route calculation is being performed, because the link-state database must remain consistent during this time. The calculation might take several seconds in a large network, and in this amount of time, many LSPs might arrive, enough to consume all credits and trigger the dropping of packets.

Once packets start to be dropped, control traffic is lost as well. In particular, hello packets might be lost, which ultimately leads to lost adjacencies and further instability.

One simpleminded way to fix this is to call `clear_if_input()` and retain the packet, which returns the credit but hangs onto the buffer. This method can lead to runaway buffer utilization, however, which makes things even worse.

The solution used by Enhanced IGRP and IS-IS is to use a secondary queue that is private to the protocol on which waiting packets are enqueued, and to limit the number of packets allowed on the queue, dropping those that do not fit. This puts an upper bound on the number of buffers that can be held by the protocol and in addition provides the opportunity to keep adjacencies alive by processing hello packets and then immediately discarding them. Additionally, Enhanced IGRP treats *any* packet received from another router as being equivalent to the last hello packet received from that system, providing a further measure of robustness.

It is worth noting that control traffic *must* have priority over all user data, even at the cost of violating traffic delivery "guarantees." If the control traffic cannot be delivered, there can be no delivery of user data.

26.3.4 Avoiding Infinite Transmit Queues and Stale Information

A commonly held belief, at least as evidenced by our code, is that the network can carry control traffic at a higher rate than the rate at which the traffic is generated. This belief is sadly mistaken, particularly because customers do silly things such as trying to cram routing updates over Frame Relay PVCs with 4 kbps of bandwidth.

In such situations, things get ugly. First of all, the information that is eventually transmitted might be stale. A route table entry might change several times in succession, so it is not helpful to transmit the intermediate states. Second, the transmit queue might pile up indefinitely, and third, in extreme cases the system might run out of memory.

The root cause of this problem is a lack of back pressure from the protocol transport to the protocol engine. Such back pressure is not necessarily trivial to implement, but it is absolutely necessary for scalability. The key is to reverse the way things are normally done. Rather than having the engine blindly generate data, the engine needs to be clocked by the protocol transport so that it generates data at the rate at which it is being transmitted, and that rate must be controllable as described above.

The key to making this work is to model the protocol as a series of state machines (typically one per interface) operating over a database. The database is updated asynchronously by events, such as incoming protocol traffic and interface state changes, and packets are built and transmitted independently. Events that update the database are no longer coupled to the transmission of information.

Link-state protocol implementations lend themselves well to this kind of treatment, because they are already organized as a database. The packets to be transmitted are simply verbatim copies of database changes.

Distance-vector protocol implementations require more thought, however, because they are not normally organized in this fashion and because the packets generated might be different on each interface. One way of organizing them is to thread the database temporally, that is, have a thread on which each entry is moved to the end as it is modified. The thread is then ordered by change time. Provide a pointer into the thread for each interface (or whatever the granularity of transmission is). In the steady state, all the interfaces point just past the end of the list. When something changes, it is moved to the end of the list and each interface state machine is started. Additional changes are also moved to the end. The information from an interface pointer to the end of the list is exactly the stuff that needs to be sent on that list. Each state machine packetizes the next bunch of information, sends it, moves the pointer, and waits for the transport to be ready again. This scheme also does the right thing if an entry changes multiple times—any interface that has not sent the previous version before it changes sends only the latest version.

26.3.5 Complexity versus Efficiency

Balancing complexity and efficiency is the heart of the engineering trade-off. At Cisco, we have traditionally done initial implementations simply, and I think that this is a good thing. It takes enough effort to get the basic functionality running reliably without adding a lot of complication. However, where we have failed repeatedly is in analyzing scalability issues and reworking the things that need to be scaled. Rather, we have waited until a crisis in the field and then rushed in the fixes.

However, the other approach—early optimism—is worse. Early optimization is one of the biggest sins of software development. It is usually difficult to determine ahead of time where the hot spots really are. Instead, the developer uses intuition to decide and often complicates code that is seldom executed.

The development and maintenance of complex software needs to include periodic performance analysis to determine where the code is going to break under stress and to make the necessary improvements without waiting for front-page stories to be printed.

There is no free lunch, however. Coding for efficiency adds complexity in return for speed. This is usually reflected in complex data structures, sometimes frighteningly so. Such complexity can be manageable, but requires much diligence in code structure and quality. As an absolute minimum, there needs to be exactly *one* piece of code that does the manipulation to create, destroy, and relink a data structure. This is, of course, good programming practice in general, but it is amazing just how many places the same bits of code pop up.

Multiple copies of this kind of code usually propagate because, in the first implementation, the operation was simple (a single pointer manipulation, for example), and the simple one-line operation was inserted directly wherever it was needed. Then things got more complicated, and the added complexity went everywhere. The obvious solution is to create a procedure to do even the most simple link, delink, allocate, and deallocate operations. (Use an inline if you do not want the overhead of the procedure call.) Then, making it more complex later is a lot easier.

Any code that does a brute-force walk of a number of entries and performs an operation on a small subset of them should be suspect. "There will never be more than 24 interfaces" was a claim that was taken to heart only a few years ago. Enough said.

26.4 Conclusion

If there is a common thread to the problems we have experienced over the years, it is that we like to ship prototypes, but then label them as production code. To some extent, the GD label has simply formalized this fact. The customers might have to wait ten maintenance releases before we trust the code enough to subject them to it by default.

This is problematic enough, but we have a history of never quite getting back to fixing the things we promised ourselves that we would fix. The apparent stability of the code helps us forget these commitments, right up until the time where the P1 bug reports start coming in.

We need to avoid quick hacks. Shipping half-baked code because it will improve time to market is almost always a false economy. The time required to do things well is seldom significantly greater than the time required to do a shoddy job, and the maintenance and rewrite overhead that comes later overwhelms any real or perceived time savings and makes the *next* product even later.

We need to "design for success," by assuming that a feature will be wildly popular and that people will build ridiculously large networks. Sometimes, it appears that we write code hoping that nobody will really use it.

We need to spend much more effort in analyzing code performance, both during development and while the code is deployed in customer networks. Providing good instrumentation enables the detection of problems early, before they snowball into network meltdowns.

Backup System

The Backup System is a redundant network connectivity scheme. One interface (network) connection takes over when the other either goes down or exceeds a traffic threshold. (This is not to be confused with Enhanced High System Availability (EHSA), which is a redundant processor scheme, in which one processor takes over when the other dies or crashes. See “Enhanced High System Availability (EHSA)” in Chapt er2, “System Initialization.”)

In 1997, the backup system was studied as part of an initiative to provide better scalability in systems with a large number of interfaces. The study yielded the decision to rewrite the system for Release 12.0, to improve not only scalability but also to improve its maintainability and performance. This chapter is an overview of the backup system, as modified for Release 12.0.

27.1 Overview

This section describes how the backup system operates and how you configure a backup interface.

27.1.1 Operation

The purpose of the backup system is to provide an auxiliary means of communication between two network devices and a definition for when this auxiliary path is used. The main interface or subinterface is known as the *primary,* and its auxiliary is known as the *standby,* or *secondar* . The primary may be a physical interface or a subinterface (as in an ATM, Frame Relay, or SMDS connection). The secondary is usually some form of dial-on-demand interface, such as a modem or switched 56K line, although other types are not precluded. Note that only physical interfaces may serve as secondaries; however, it does not make sense to use subinterfaces as standbys.

The backup system, as currently implemented, provides two mechanisms whereby the standby may be made active. The first is an ordinary backup mechanism. When traffic to a foreign network is of high importance and the primary link goes down, then the router may be configured to bring up a standby, which may also serve to route packets to the remote network. When the primary returns to its operational state, the secondary may then be returned to the standby state. Activation and deactivation of the secondary may occur at once, after a specified delay, or may be disabled altogether in this mechanism.

The second mechanism is useful in eliminating bottlenecks in the network, for it makes the standby active when the network load on the primary interface exceeds a given threshold. The secondary is also deactivated when the load drops back down below another given threshold. Load triggers may also be disabled.

27.1.2 Configuring Interfaces

This section details how the interfaces are configured:

- Specifying the Standby Interface
- Specifying Backup Delays
- Specifying Backup Loads, Main Interfaces Only

27.1.3 Specifying the Standby Interface

To enter configuration mode, your router must be in the *enabled* state. Once in enabled mode, type **configure terminal**. After specifying the primary interface/subinterface (the interface to be backed up), use the **backup interface** command to specify the secondary interface:

```
router# configure terminal
Enter configuration commands, one per line.  End with CNTL/Z.

router(config)# interface serial 0
router(config-if)# backup interface serial 1
```

or:

```
router(config)# interface serial 0.1
router(config-subi)# backup interface serial 1
```

The above commands specify that the Serial 1 interface is to be used as a standby for the Serial 0 interface, or for the Serial 0.1 interface in the second case.

To unconfigure an interface from being backed up, specify **no backup interface**. This sets any other backup settings, such as backup delays and backup loads (described below), back to their default settings.

27.1.4 Specifying Backup Delays

By default, there are no backup delays unless defined. This means that if a primary goes down, the secondary is immediately brought up. It also means that if the primary comes back up, the secondary is immediately put back into standby mode. Delays offset the transitions in time and may be set as follows (in seconds):

```
router(config-if)# backup delay 5 10
or:
router(config-if)# backup delay 5 never
or:
router(config-if)# backup delay never 10
```

In the first case, we're saying that we want the secondary to come up after the primary has been down continuously for five seconds, and that we want the secondary to go back to standby mode after the primary has been up continuously for ten seconds.

In the second case, we want the secondary to come up after the primary has been down continuously for five seconds, but we never want it to be put back into standby mode. It should remain up.

In the third case, we have specified that the secondary is never to come up if the primary fails, but we still want it to be brought down after the primary has been up continuously for ten seconds. This might be useful in the case where the secondary is already active but it was decided that it would not be allowed to come up again. The use of this third form is somewhat questionable, but it is available for use since it is already out in the field.

The fourth case, not listed above, is `backup delay never neve` . This is disallowed since it disrupts operation of the backup system. It is probably not useful, either.

If backup delays are left unspecified, the default is `backup delay 0 0`. Backup delays may be returned to their default by explicitly setting **backup delay 0 0** or **no backup dela** .

27.1.5 Specifying Backup Loads, Main Interfaces Only

The backup load mechanism is specified in terms of percentages of the possible load of the *main* interface, in the form of numbers from 1 to 100.

Note This means that the **backup load** command is not available on subinterfaces.

The same **never** keywords are accepted here:

```
router(config-if)# backup load 70 50
or:
router(config-if)# backup load 70 never
or:
router(config-if)# backup load never 50
or:
router(config-if)# backup load never never
```

In the first case, when the backup load exceeds 70% of the available bandwidth of Serial 0, the secondary, Serial 1, will be brought up. When the load drops back below 50% of the available bandwidth of Serial 0, the secondary will be returned to standby mode.

In the second case, the secondary will go up after the load is exceeded, but it will not be disabled when the load drops back down.

In the third case, the secondary will not go up after the load is exceeded, but it will be brought down after the load drops back down.

To unconfigure and disable the standby from being affected by load transitions, **backup load never never** may be specified, and since it is the default, **no backup load** achieves the same result.

27.1.6 Notes On Operation

In this section, some fine points are addressed that concern a few peculiarities of the backup system.

First is the case where both a primary interface and one or more of its subinterfaces are being backed up. In this scenario, if the main interface goes down, the backup for the subinterface is not activated; instead, the backup for the main interface is activated in its stead. If only the subinterface goes down, though, then the backup for the subinterface is activated, as it would be if its main interface had not been backed up. This is to prevent double-backups from occurring needlessly.

Second is the case where an interface that has been backed up due to an overload situation then goes down. This is known as the backup/overload situation. The secondary interface is not brought back to standby mode until two things happen: the primary interface has come back up and the load on the primary has dropped back below the given load threshold.

27.2 Description of Changes

The pre-12.0 backup code contained a number of problems that made it a prime candidate for a rewrite. The changes made are documented in this section.

Problem

Passive timers were being polled to detect timeouts. Polling occurred on all hardware IDBs and all subinterface software IDBs off of each hardware IDB, once per second.

Solution

Passive timers were replaced by managed timers, and an existing background service routine was modified to service the new (managed) backup timers.

Problem

All interfaces were being checked on a periodic basis (currently five seconds) to determine if backup loads needed to be calculated.

Solution

From information provided by the runtime configuration, a private IDB list was constructed to include all interfaces which required backup load calculations. This list is traversed during the normal five second interval, instead of scanning all IDBs.

Problem

All subinterfaces were being checked once per second to determine state changes, as there is not currently a registry call in place which is invoked when subinterfaces change state.

Solution

Again, a private IDB list was constructed from the runtime configuration which contained a list of all subinterfaces which required scanning. This list is traversed once per second. (A better solution would be to implement a subinterface statechange call and instrument it throughout the code wherever subinterface statechanges are made. Then, a callback routine could be registered by the backup system to detect and act on these state changes. However, this solution is not in place at the current time).

Problem

Backup-related timers and parameters were stored in every hardware IDB and software IDB.

Solution

Almost all backup-related timers and paramters were moved into a newly created subblock type, the backup subblock. Backup subblocks were allocated solely to interfaces which were being backed up (primaries) or operated as standby's (secondaries).

Problem

For standby interfaces, determining the primary interface was very difficult, and required running through each and every hardware and software IDB.

Solution

The subblock contains pointers to both an interface's primary interface (if it is a standby) and its standby interface (if it is a primary). This bidirectional link makes coding much simpler and more efficient.

Problem

The addition of subinterfaces to existing backup code created code duplication (one set for main interfaces and one set for subinterfaces). This made the code difficult to follow and hard to maintain. It provided an environment to possibly make changes inconsistently between the main interfaces and the subinterfaces.

Solution

The code was unified to make use of the subblocks instead of hardware and software IDBs. Thus, main interfaces and subinterfaces use exactly the same code.

Problem

Backup code was spread throughout many different procedures and was very difficult to maintain.

Solution

Backup code was modularized out and placed into its own module. Backup operation was converted from distinct calls located in *many* places to a finite-state-machine with event notification. Only a select few calls were made available for use outside of the backup system.

Problem

Backup code could not be easily pulled out of the mainline code.

Solution

Since the new backup code was made modular, it was turned into a subsystem. Backup initialization code was called as part of normal subsystem initialization. A new registry was created to contain the backup registry calls. Existing parser commands were taken out of the parser chain and reintegrated as part of the backup subsystem initialization in the form of a parser extension request.

Problem

There was no way to debug backup events.

Solution

A “debug backup” command was added to help debug backup events.

Problem

There was no way to view backup states.

Solution

A “show backup” command was added to show backup states.

CHAPTER 28

Verifying Cisco IOS Modular Images

This chapter was originally Cisco IOS Technical Note #4, written February 27, 1997. It was moved to this chapter in September 1998.

A modular image is a collection of linked object files that contain no unresolved references. This chapter explains how verify the Cisco IOS source code modularity using modular image targets in the `sys/makefile` files and in the platform-specific makefiles.

28.1 What is a Modular Image?

A *modular image* is a collection of linked object files that contain no unresolved references. The object files that you choose to collect into a modular image are files that form a logical subset of the Cisco IOS software.

Building the modular images on a regular basis allows an automated check of the degree to which Cisco IOS programmers are respecting the existing modularity of the Cisco IOS code base.

What a Modular Image Is Not

A modular image is not necessarily intended to run. It it merely intended to be an isolated set of files that have no external references.

A modular image does not in and of itself implement software modularity.

A modular image is not a substitute for creating application programming interfaces (APIs) or application binary interfaces (ABIs).

28.2 Why Create Modular Images?

The main purpose of creating and building modular images is to verify that a logical subset of the code that has been identified as a module is self-contained and has no unresolved references. You verify the modularity of an image when you first define the module and then on an ongoing basis to ensure that continuing work on the Cisco IOS code has not broken the modularity. Also, as part of the standard build process, existing module images are checked nightly and built weekly to verify that recent changes to the code have not broken the modularity.

28.3 Types of Modularity Checks

You can verify Cisco IOS modular images in two ways. Both of these methods are implemented in the `sys/makefile` files and platform-specific `makefiles`.

- Build modular images successfully, without link errors. You do this by running `make` using the `sys/makefile` files and platform-specific `makefiles`. In the `sys/makefile` files, you use the targets `modular.all` and `modularity_check.all` to build modular images. In the platform-specific `makefiles`, you use the targets `modular` and `modularity_check`.
- Run the `sys/scripts/connect` Perl script. This script takes a list of object files from `STDIN` and checks them for unresolved references. For help about the usage of this script, enter the command `connect -h`.

When you are developing Cisco IOS code, you are strongly encouraged to check the modular images for the platform on which you are developing before committing the changes. You do this especially if your code references data or functions in more than one subsystem.

28.4 Modularity Targets

In the `sys/makefile` files, the modularity targets are `modular.all` and `modularity_check.all`. In the platform-specific `makefiles`, the modularity targets are `modular` and `modularity_check`.

You should never modify the list of modular images in the modular target. If you add a new feature to the Cisco IOS code, add its files to the list of modular images.

28.5 Build Modular Images for a Single Platform

You can build all the modular images for a single hardware platform with the modular target in the platform-specific object directory. This target builds the complete set of modular images defined in the `sys/makeimages` file for that platform.

28.5.1 Build All Modular Images for a Single Platform

To build all the modular images for a single hardware platform, follow these steps:

Step Change into the platform directory:

```
cd sys/obj-processor-platform
```

Step Build the modular images:

```
make -k modular >& log_file
```

The `-k` option, known as the "keep-going" option, permits the `modular` rule to continue even if a particular modular image fails to link.

`log_file` is the name of a log file. This file will contain the output of the `make` command, including any error messages.

Each modular image built in Step2 is deleted after it is created; it is never written to the `/tftpboot` directory. Any errors that occur during the build process are recorded in the log file. You should resolve all errors until the image builds successfully. When resolving errors, you should attempt to maintain the smallest possible modular image. Do not add other subsystems to the modular image if you can avoid it.

28.5.2 Build a Specific Modular Image for a Single Platfor

To build a specific modular image for a single hardware platform, follow these steps:

Step Change into the platform directory:

```
cd obj-processor-platform
```

Step Build the modular images:

```
make -k modular-modularity_type >& log_file
```

The `-k` option, known as the "keep-going" option, permits the `modular` rule to continue even if a particular modular image fails to link.

`modularity_type` is the defined logical grouping of functionality whose modularity you want to check. The `modular-`*`modularity_type`* targets are defined in the variable `MODULAR` in the file `sys/makeimages`. Currently, the variable contains the following targets:

```
MODULAR = modular-apollo modular-at modular-ataurp modular-atip modular-clns \
modular-dialer modular-dn modular-fr modular-fr-svc modular-ip \
modular-ipx modular-ipxeigrp modular-ipxwan modular-mop \
modular-nlsp modular-smds modular-snapshot modular-snmp modular-sntp\
modular-tb modular-tiny modular-ukernel modular-vax modular-vines \
modular-x25 modular-xns
```

`log_file` is the name of a log file. This file will contain the output of the `make` command, including any error messages.

Each modular image built in Step2 is deleted after it is created; it is never written to the `/tftpboot` directory. Any errors that occur during the build process are recorded in the log file. You should resolve all errors until the image builds successfully. When resolving errors, you should attempt to maintain the smallest possible modular image. Do not add other subsystems to the modular image if you can avoid it.

28.6 Build Modular Images for All Platforms

You can build all the modularity images for all Cisco IOS platforms with the `modular.all` target in the `sys/makefile` file. The `modular.all` target compiles all the objects necessary to link the modular images. Running this target can take from approximately 3 hours to over 12 hours, depending on how many objects need to be compiled, the performance of your compile server, and the number of platforms and modular images defined in the `sys/makeimages` file.

To build all the modular images for all hardware platforms, follow these steps:

Step Change into the `sys` directory:

```
cd sys
```

Step Build the modular images:

```
make -k modular.all >& log_file
```

The `-k` option, known as the "keep-going" option, permits the `modular` rule to continue even if a particular modular image fails to link.

`log_file` is the name of a log file. This file will contain the output of the `make` command, including any error messages.

Each modular image built in Step2 is deleted after it is created; it is never written to the `/tftpboot` directory. Any errors that occur during the build process are recorded in the log file. You should resolve all errors until the image builds successfully. When resolving errors, you should attempt to maintain the smallest possible modular image. Do not add other subsystems to the modular image if you can avoid it.

28.7 Check Modularity with the sys/scripts/connect Script

If you have already compiled all the object files in a modular image, you can quickly check for modularity breaks using the `modularity_check.all` target, which is defined in the `sys/makefile` file. This target calls the Perl script `sys/scripts/connect`, which checks for unresolved references in the object files in the modular images.

There are two advantages to using the `modularity_check` target: it completes the check of all modular images for all platforms in about one hour, and it provides a list of all unresolved external references for all files in each image.

The `modularity_check` target does not check that all objects required to build a modular image are present, and it does not attempt to rebuild those objects before it runs `sys/scripts/connect`. This saves a significant amount of time. If a required object file is missing, the script prints an error message indicating this.

Note We have observed that for images built with the MIPS linker, unresolved references are left in the image and they are called out as modularity breaks. These unresolved references occur both in the modular and production images. Currently, we are investigating this anomaly to determine whether the unresolved references indicate a bug in the linker or a bug in the utilities (`nm` and `objdump`) that read the unresolved references from the image files.

28.8 Modularity Checking Done by the Nightly Builds

Starting with Release 11.2, the nightly build of Cisco IOS production images runs the `modularity_check.all` target to verify the modularity of all modular images. Also, once a week, the nightly builds build the modular images using the `modular.all` target. The modular images are not archived, but are removed immediately after they are built.

Reports of nightly build failures, which include modularity breaks, are mailed to the alias `nightly-build-failures`. A modularity break is defined as an unresolved reference in the modular image found either by the linker or the `sys/scripts/connect` script. If a modularity break occurs, a DDTS bug report is filed against the development group with responsibility for the software.

The build group posts all nightly build results for a given release to the newsgroup `cisco.eng.nightly.`*`release_abbreviation`*`-build`. The nightly build newsgroup for the California release is `cisco.eng.nightly.cal-build`.

CHAPTER 29

Writing DDTS Release-Note Enclosures

This chapter was originally Cisco IOS Technical Note #3, which was written in January 1997. It was moved to this chapter in September 1998.

The text in a DDTS release-note enclosure describes a problem reported in DDTS to customers. This chapter provides guidelines for writing release-note enclosures. It assumes that you are familiar with DDTS. For information about DDTS, see the "Getting Help" section of this chapter.

29.1 What Is a Release-Note Enclosure

A *release-note enclosure* is the enclosure in a DDTS bug report that describes the problem to Cisco customers and partners. (Note that enclosures are sometimes also called *attachments*.) The purpose of release-note enclosures is to provide timely, accurate, and useful information about actual and potential problems with Cisco software, hardware, and documentation products. The description in a release-note enclosure should allow the customer to identify the problem and should provide a workaround if one is known.

To describe the problem in the bug report to internal Cisco people, you use other DDTS enclosures, such as the Description enclosure.

29.2 How Customers See Release-Note Enclosure

Customers see release-note enclosures in one of the following ways:

- Cisco Connection Online (CCO). The CCO Bug Navigator, which is part of the BugToolKit, allows customers to view the headline, release-note enclosure, and other information about a DDTS bug report. All registered CCO users can view all bug reports that have release-note enclosures, regardless of the state the bug report is in, with the following exceptions:
  - Bugs with release-note enclosure text whose first line is $$IGNORE
  - Bugs with no release-note enclosure; these are visible only to internal users and Cisco partners

  The CCO user interface indicates whether a bug has been junked (state J), is a duplicate (state D) of another bug, or cannot be reproduced (state U).
- Cisco Connection Documentation (CCD, formerly UniverCD). This is a monthly CD-ROM produced by the Knowledge Products group. For each currently supported Cisco IOS software release, CCD contains a file that lists all the release-note enclosures.

- Software release notes. These contains the release-note enclosures for catastrophic and severe problems reported in the DDTS database. Printed copies of the release notes ship with all software shipments. The release notes are also included on CCD.

When a bug report has a release-note enclosure, the report is also distributed to Cisco's field personnel and various business partners via e-mail and anonymous FTP.

Release-note enclosures for bug reports about software that is in the external verification phase are visible only in electronic form and only to internal Cisco audiences and external verification sites that have access to CCO.

29.3 Who Writes Release-Note Enclosures

The initial release-note enclosure is written by the person who submits the bug report. Development Engineers (DEs) and Customer Engineers (CEs) can add information to or modify the information in a release-note enclosure. For details about writing responsibilities, see the "DDTS Release-note Enclosure Process" web page.

29.4 When Do Release-Note Enclosures Get Written

You write a release-note enclosure when you first submit a bug report. You or others can revise it at any time.

29.5 Writing Release-Note Enclosures

This section discusses the issues involved in writing release-note enclosures.

29.5.1 Naming a Release-Note Enclosure

A release-note enclosure is an enclosure in a DDTS bug report that has the following title:

Release-note

Note the exact spelling and capitalization. Deviations from this exact title can cause DDTS scripts to fail.

29.5.2 Writing Guidelines

Release-note enclosures must include enough information so that the customer can recognize the problem and, if possible, implement a temporary workaround or permanent solution. A release-note enclosure should include the following type of information:

- Conditions Under Which the Problem Occurs
- Symptoms of the problem
- Workaround or solution, if any

In the release-note enclosure, describes the problem as it exists, even if it has already been fixed. This is because the problem might not have been fixed in all releases (if a DDTS bug report applies to several software releases), or the fix might not have been integrated into all releases.

Remember that the audience for release-note enclosures is Cisco customers.

29.5.2.1 Conditions Under Which the Problem Occurs

The conditions describe the customer environment under which the problem has occurred or might occur. Include the following information if relevant:

- Hardware configuration. If the problem affects only specific hardware versions, state this explicitly.
- Software configuration. If the problem affects only specific software releases, state this explicitly.
- Router configuration commands that cause the problem.
- Problem frequency. State whether the problem always occurs under the stated conditions, whether it occurs occasionally, or whether it occurs only infrequently. If the problem is infrequent or if there is only a low probability that a customer might encounter the problem, say this explicitly by stating, "Under rare conditions..."You do not want to alarm customers unnecessarily.

Example

On Cisco 4000 series routers running Release 10.3(4), ...

29.5.2.2 Symptoms

The symptom is a clear, brief description of the problem. This description should allow the customer to match the problem to something they might see on their device.

Example

If the **source-bridge proxy-explorer** command is configured, a Token Ring interface might intermittently not receive packets.

29.5.2.3 Workaround

If a temporary or permanent workaround or a permanent solution to the problem is known, describe it. If there are any limitations caused by the workaround or solution, state them.

Example

The workaround is to turn off proxy explorer. One side effect of doing this is that explorer traffic on the network will increase.

29.5.3 Writing Style

When writing release-note enclosures, following these writing-style guidelines:

- Write in present tense.

  DO—If a serial interface is set to loopback via a hardware signal, the interface remains in loopback until the hardware signal is dropped.

  DON'T—If a serial interface is set to loopback via a hardware signal, the interface will remain in loopback until the hardware signal is dropped.

- Write in active mood (active voice).

  DO—If you configure secondary addresses on an interface that you have otherwise configured as unnumbered, the interface routes corresponding to these addresses are not advertised in IS-IS.

  DON'T—If secondary addresses are configured on an interface that is otherwise configured unnumbered, the interface routes corresponding to these addresses are not advertised in IS-IS.

- Keep the problem description as short as possible. Do not include unnecessary information.
- Write in complete sentences.

  DO—Cisco 2500 series routers might reload with a bus error at PC 0x30E9A8C.

  DON'T—2500 Router reloaded with bus error at PC 0x30E9A8C for unknown reason.

- Use complete Cisco product names, such as *Cisco 10005 router*, *Cisco 70000 series routers*, or *AS5100 access serve* . Do not use abbreviations, such as *c7000* or *7000 series*, or internal code names, such as *Volcano*.
- Refer to specific releases of the Cis coIOS software as *Cisco IOS Release x.y(z)* or simply *Release x.y(z)*. Do not use *version x.y(z)* or *x.y(z)*.
- When referring to the Cisco IOS software, use *Cisco IOS* as an adjective. For example, say *Cisco IOS code*, not *IOS code* or *IOS*. This is necessary to protect our trademark of "Cisco IOS."
- Avoid unnecessary or irrelevant comments that do not add useful information. For example, avoid the following types of comments:
  - This is a weird bug.
  - This is a new router
  - The customer upgraded.
  - The customer is very upset.
- Avoid using slang, jargon, and internal code names that the customer might not understand. For example, avoid the following terms:
  - Crash
    As alternatives, use *reload* (verb), *system reload* (noun), *unexpected system reload* (noun)
  - Hang
    As alternatives, use *pause indefinitely*, *stop*, or *stop working*
  - Bug
    As alternatives, use *problem*, *possibly unexpected behavior*, or *aspect of the implementation*
  - Brain-dead design, stupid design, users stupid enough to do
    Rewrite to omit these phrases
- Do not include any angle brackets (< and >) in the text of a release-note enclosure except for character formatting. These break the scripts that gather the release-note enclosures.

29.5.4 Text Formatting Guidelines

29.5.4.1 Character Formatting Guidelines

Some scripts gather the release-note enclosure text for inclusion in formatted documents, such as FrameMaker documents. Toproperly identify commands, command arguments, and command keywords, include character formatting strings in the release-note enclosure text, as follows:

- Use bold for command names and command arguments.
- Use italics for command keywords.
- Use italics for any words that you want to emphasize, such as the word *no*.

Do *not* place single or double quotation marks around command name.

Table 29-1 explains the character formatting strings. Note that these strings are *not* case-sensitive.

Table 29-1 Character Formatting Strings

Tag	Function	Example	End Result in Formatted Document
<CmdBold>	Marks the beginning of a command or command argument	<CmdBold> dialer string<NoCmdBold>	**dialer string**
<NoCmdBold>	Marks the end of a command or command argument		
<CmdArg>	Marks the beginning of a command keyword	<CmdArg>dialer-string<NoCmdArg>	*dialer-string*
<NoCmdArg>	Marks the end of a command keyword		

Do not nest formatting strings within other strings. Nested strings are ignored.

DO—<CmdBold>route-map<NoCmdBold> <CmdArg>map-tag<NoCmdArg> <CmdBold>permit<NoCmdBold>

DON'T—<CmdBold>route-map <CmdArg>map-tag <CmdBold>permit<NoCmdBold>

Examples: Character Formatting Guidelines

The following is an example of release-note enclosure text that includes formatting tags:

> If you use the <CmdBold>dialer string<NoCmdBold> <CmdArg>dial-string<NoCmdArg> command on an ISDN interface instead of a <CmdBold>dialer map<NoCmdBold> command, the router might crash.

In formatted documents, this text appears as follows:

> If you use the **dialer string** *dial-string* command on an ISDN interface instead of a **dialer map** command, the router might crash.

29.5.4.2 Other Formatting Guidelines

Certain characters interfere with the scripts that process release-note enclosures. So far, the only characters we know that cause problems are the angle brackets (< and >) when they are used for purposes other than character formatting. Follow these guidelines to avoid these problems:

- Spell out the terms *greater than* and *less than.*
- Do not enclose command arguments in angle brackets.
- Do not enclose variables that might appear in error messages in angle brackets.
- Remove angle brackets that might by displayed in error messages.

29.5.5 Guidelines for Using $$IGNORE in Release-Note Enclosures

DDTS bug reports that identify problems that should not be seen by customers should not have release-note enclosures.We do not want customers to see the following kinds of problems:

- Problems that affect an internal structure or operation of the code that is not visible to the user Examples include problems in the Cisco IOS kernel, such as the scheduler or managed timers, and code restructuring.
- DDTS bug reports for makefile or other changes that affect how images are compiled.
- Problems that might involve sensitive competitive information.

In all these cases, you must make a deliberate decision not to document the problem.

To prevent a bug report from being seen by customers, create a release-note enclosure that contains the following text at the beginning of the enclosure:

```
$$IGNORE
```

If you use the $$IGNORE string in the release-note enclosure, the entire bug report is not visible to customers. However, it is still visible to internal users and to Cisco partners.

Make sure there are no spaces or blank lines before the $$IGNORE string. Any blank lines or other text before this string will *not* prevent the DDTS bug report from being visible to customers.

The $$IGNORE string is case-sensitive, so make sure that you type it exactly as shown.

In the remainder of the release-note enclosure, you can explain why the bug report is not being documented and who decided not to document it.

Do not use the $$IGNORE string if you do not have enough information to describe the problem to the customer or if you plan to write a release-note enclosure at a later date. If you have insufficient information, simply leave the bug report without any release-note enclosure. Doing this allows tools to distinguish between bugs that still need to be documented and those that are deliberately undocumented.

29.5.6 Sample Release-Note Enclosures

The following are examples of good release-note enclosures:

- The router may reload when trying to execute the <CmdBold>show accounting<NoCmdBold> command.
- QLLC cannot use X.25 PVCs for DLSw+. The workaround is to use RSRB or to use X.25 SVCs.

- When multiprotocol traffic such asIP, DECnet, XNS, AppleTalk, and IPX is passed to a Cisco 2500 or Cisco 4500 router through a Token Ring interface, the router cannot accept all the traffic. This sometimes results in the Token Ring interface being reset and packets being dropped.
- On Cisco 7500 RSP platforms, FSIP serial interfaces may display the following panic messages on the RSP console.
  %RSP-3-IP_PANIC: Panic: Serial12/2 800003E8 00000120 0000800D 0000534C
  %DBUS-3-CXBUSERR: Slot 12, CBus Error
  %RSP-3-RESTART: cbus comple
  If the string "0000800D" is included in on the panic message, the problem is related to this bug. The workaround is to load a new image that contains the fix for this bug.

The following examples show inappropriate release-note enclosures and provide examples for rewriting them:

- RSP2 reload at rsp_ipfastswitch

  PROBLEM Incomplete sentence; reference to internal software routine.

  SUGGESTED REWRITE: RSP2 systems might reload while performing RSP fast switching.

- --- Release-note ---
  All SNA traffic with local-ack while using reverse sdllc (rsdllc) feature will fail There is no known workaround, however a code fix has been identified and tested successfully. The fix will be available in 11.2(3.1) and 11.2(4).
  engineer's name and date
  Another test was executed on 12/20/96 with images built out of the latest California branch. And it was successful (with no code change applied). So it seems the problem was fixed in the latest California branch unknownly. And no code change needs to be applied. *engineer's name and date*

  PROBLEM Do not type the name of the enclosure— --- *Release-note* --- — in the text. Do not include your name or the date. Do not mention when a fix might be available. Do not include test information.

  SUGGESTED REWRITE: All SNA traffic that uses local-ack and reverse SDLLC fails. There is no known workaround.

- After receipt of "rogue" XID3 from its partner, DSPU may become stuck in XID state; and therefore, connection will never become active.
  Work-around is to stop and re-start the DSPU connection via "no dspu start"/"dspu start" configuration commands

  PROBLEM Incomplete sentence; placing commands in quotes.

  SUGGESTED REWRITE: After receipt of a "rogue" XID3 from its partner, DSPU might become stuck in XID state and as a result, the connection will never become active. The workaround is to stop and then restart the DSPU connection using the <CmdBold>no dspu start<NoCmdBold> and <CmdBold>dspu start<NoCmdBold> commands.

29.6 Writing DDTS Headlines

Although not part of the release-note enclosure, the DDTS headline is visible to Cisco customers partners regardless of whether there is a release-note enclosure. Follow these guidelines when writing DDTS headlines:

- Write a concise description of the problem symptom. The headline can be up to 65 characters long.
- Do not include references to unreleased products.

- Do not refer to released products by their internal project names.
- Do not use offensive language.

- Do not refer to source code routines.
- Avoid references to other DDTS reports.

29.7 Getting Help

If you need help writing a well-composed release-note enclosure, try to find someone in your group who you think is a good writer and have them help you. If you are working with writers from Knowledge Products, ask if they could help you. If neither of these options works, ask the manager of the Engineering Education group if a writer or editor from the group can help.

For additional information about writing release-note enclosures, see the "DDTS Release-note Enclosure Process" Web page.

For information about using DDTS, see the *Cisco Engineering Tools Guide* .

To enroll in the DDTS course, go to the "Cisco Engineering Training" Web page.

Appendixes

APPENDIX A

Writing Cisco IOS Code: Style Issues

A.1 Purpose of This Chapter

Frequently, a newcomer to the Cisco IOS software engineering group is upbraided for not doing something "the right way," and the newcomer will note something to the effect that "if someone had provided useful documentation, writing Cisco IOS code would be much easier."

The purpose of this appendix is to document The Right Way™ to write Cisco IOS code. This appendix addresses the issues and conventions of the Cisco IOS software group. It does not address issues of other Cisco software groups, such as the microcode group.

A.1.1 Coding Conventions: Something for Everyone to Protest

The Cisco IOS software, despite popular misconceptions to the contrary, has conventions for designing, writing, and documenting code. However, the rapid growth of the software engineering community at Cisco has outstripped our earlier method of communicating these coding conventions to newly hired engineers and engineers in newly acquired companies. Previously, experienced engineers passed on the conventions, designs, technology, and wisdom in "nerd lunch" talks. However, with Cisco's rapid growth and the wholesale assimilation of engineering groups from acquired companies, which are often no longer geographically co-located with the experienced engineers, the nerd lunch training method no longer scales. As a result, the Cisco IOS code base is growing into a mishmash of conflicting conventions and methodologies.

The purpose of coding conventions is quite simple: to facilitate the rapid understanding of any piece of Cisco IOS source code by any engineer. This results in clearer expression of engineering intent, fewer bugs, less time spent training engineers, and engineers being able to move from project to project with greater ease.

Some of the issues addressed in this appendix are magnets for controversy, especially topics such as pretty printing and white space conventions. Engineering practices that get a more reliable product to market more quickly can be directly translated into the tangible benefits of larger market share and higher revenues for the company as a whole. For you, the engineer, this translates into higher stock prices. And if you want to publicly claim to other Cisco employees that you do not care about making the stock price go up, you might as well smear yourself in A-1 Steak Sauce and jump into the polar bear exhibit in the Anchorage, Alaska, zoo. You'll live longer in the exhibit with the bears.

The foregoing rationale for coding conventions might not be enough for some readers, who might offer protestations that their personal conventions, used for much, if not most, of their careers in software engineering previous to their employment at Cisco, are technically superior. And some of these arguments may well be correct. But a major goal in coding is consistency of coding style and implementation. Whether you think Cisco's conventions are good, bad, or indifferent, the current

coding style is the one we chose more than 10 years ago. Everyone who has joined the company since then has had to conform to it. Unless there is an overriding reason to change (read: "We sell more product, resulting in higher stock prices"), you will also have to conform.

Because it is impossible to arrive at agreement about every coding convention issue, all engineers might find something in this document that is not to their ultimate liking. However, you are reminded that in a successful compromise, everyone feels equally shortchanged.

A.1.2 Definitions

The following terms are used throughout this appendix:

Platform-dependent code: Code that has real and intimate knowledge of the platform or device that it controls. Typical examples are the bootstrap code, device drivers, and Flash memory drivers. Also typically included in platform-dependent code is the fast-switching code, because the ultimate performance in the packet fast-path depends on very specific use of the platform's hardware.

Platform-independent code: Code that does not care or specifically know which platform is running it. Examples include routing protocols, the scheduler, the error logger, and other high-level features.

Device drivers: Code used to control interface cards or specific chip sets in interface cards.

A.1.3 What This Appendix Addresses

This appendix addresses a number of higher-level issues in writing Cisco IOS software. These issues affect the integration of your individual code with other code in the Cisco IOS software engineering community. This appendix discusses the following topics:

- Design Issues
- Using C in the Cisco IOS Source Code
- Presentation of the Cisco IOS Source Code
- Variable and Storage Persistence, Scope, and Naming
- Coding for Reliability
- Coding for Performance

A.1.4 What This Appendix Does Not Address

No document about coding styles presented to an engineering audience the size of Cisco's can hope to cover every topic. As such, there are some issues that this appendix explicitly does not cover and does not intend to cover in the future, including the following:

- Debugging a specific issue or problem (See Chapter18 , "Debugging and Error Logging.")
- Debugging errors returned by the tool chain

A.2 Design Issues

Before you add large new pieces of functionality and code to the Cisco IOS source, you should design them to use and fit within the Cisco IOS architecture. Although the architecture is often a moving target, your features and code should attempt to adopt the latest available interfaces, data

structures, and libraries. The Cisco IOS source contains a large amount of legacy code. If you add new features to the Cisco IOS code that are implemented using old and possibly deprecated architecture, you are adding to the work of bringing the legacy code forward into the new architecture. You might think you are saving time, but it will likely cost you and the company more time and money to re-engineer the new features into the new architecture than it will take you to understand and use the new architecture in the first place.

There are several broad categories of design issues that are germane to all projects. The following sections are not meant to be all-inclusive but rather guidelines for design issues that, if addressed early in the implementation of your Cisco IOS features, will make your job considerably easier.

A.2.1 Do Not Use Conditional Compilation for Platform-Specific Code

The Cisco IOS source used to be very different than what you see now. In the versions of the Cisco IOS software before Release 9.21, little code was platform-independent. This was because much of the code was littered with conditional compilation statements similar to the following:

```
#ifdef PAN
/* Logic for “Pancake” or IGS/C3000 platforms */
#else
#ifdef CBUS
/* Logic for AGS+ platforms */
#else
/* Logic for “High-end” or AGS/MGS/CGS platforms */
#endif
#endif
```

As a result of this coding practice, errors found and fixed on one platform might not be fixed on another, little object code was shared even among platforms using the same CPU, and porting the Cisco IOS software to a new platform was very difficult because the platform-dependent issues were spread throughout the code, not neatly encapsulated in well-defined places with well-documented interfaces.

As tempting as it might be to add conditional compilation for a specific platform, device, interface or chip, don’t do it. Take the extra time to separate generic code from platform-specific or device-specific code, and take the time either to use an existing interface between generic and nongeneric code or to develop a new interface.

A.2.2 Plan Your Feature as a Subsystem

When you are adding a new feature to the Cisco IOS code, it is very likely that it is something that is not absolutely essential for the Cisco IOS system to run. In other words, there might be some customers who are never going to use your feature and who would be rather irate at having to fill their router memory with features that they will never use.

Currently, the only way to selectively omit features from an image is to write features in their own subsystems and to use the registry mechanism to create bindings at system boot-time between your feature’s code and the rest of the system. Again, you might be tempted to use conditional compilation to control whether your feature is included in a particular image, but as with platform-specific conditional compilation, if everyone does it, no one can maintain it. There are many examples throughout the source of how to effectively and easily design a feature as an optional part of the system. Two good examples to cite here are the AppleTalk and VINES subsystems.

A.2.3 Do Not Overload Existing or System Registries

In the past, there was a tendency to put new feature registry points into existing registry definitions. This might have been convenient, but it was not germane to the requirements of the new feature's interface.

An example of putting new registry points into existing registry definitions would be putting a protocol-specific interface point (for example, AppleTalk Echo packet reception) into a systemwide interface registry, which is the generic interface registry for all hardware interfaces. Although this works and the new functionality might well be modular, the pollution of the generic interface definition creates problems when porting the Cisco IOS code to new platforms. Also, adding a registry point in a nonobvious place makes it hard to document and maintain the functionality. It is better to create a new registry than "pollute" existing registries if your feature's interfaces are not logically associated with the existing registry.

However, do not create new registries gratuitously. Interface design and specification require careful thought. Do the design work necessary to create easily understood and maintained registries.

A.2.4 Don't Be a Stub Slob; Use Registries

Registries allow portions of the Cisco IOS code to be isolated from other portions of the code while maintaining the modularity of the code. For example, you use registries to create platform-specific functions.

Before registries were created, Cisco IOS code was made "portable" between various target platforms through conditional compilation and a rather festerous technique called "stub" functions. Stub functions were essentially functions with the same name as a function that had a real purpose on one platform but no purpose on other platforms. Typically, these stub functions were called via function vectors in structures or tables. If omission of the function entry point on a platform where the real function was not needed would cause a link error, a "stub" function was introduced into the source code to allow the image to link.

The correct technique to provide for optional functions on a per-platform basis is to use registries. Moreover, you should think of registries as generic interface points between functional layers or modules, not feature-specific or function-specific entry points. You should rarely, if ever, create new stub registries. For a good example of how a registry is created, see `ip/ipfast.c`. This registry correctly populates the IP fast-switching cache independently of the type of hardware assistance that might be available to use for fast switching.

A.2.5 Don't Hog the Chip

It goes without saying that everyone writing code for the router would prefer that their code were the only code that the CPU were running. Alas, this isn't the case, and you must share the resource. The Cisco IOS code does not force you to share the CPU. Rather, it is incumbent upon every process running on the router to surrender control of the CPU at frequent intervals to allow other processes to get their share of the CPU. This is what is referred to as *cooperative multitasking*, and in the Cisco IOS code it operates much the same as Windows 3.1 and Macintosh systems.

The benefits of cooperative multitasking are that the amount of CPU used by the Cisco IOS code to schedule processes is much less than would be used by a preemptive multitasking scheduler. Also, the latency in handling real-time events is lower, because you can design a process to handle small, well-defined tasks quickly and without preemption. The downside of using a scheduler without preemption is that each process and subsystem must be designed carefully to ensure that the whole system runs smoothly.

To make your job of implementing and supporting your features easier, keep the following in mind when designing features:

- Design potentially CPU-intensive tasks so that they consist of small, atomic fragments of work that lend themselves to checkpointing and process suspension. This type of design is very difficult to add after your feature has been written, because checkpointing and frequent surrender of the CPU require more complex data structures.
- Handle events that must happen at appointed times—for example, generating and transmitting of routing keepalive packets—in their own process, not by grating the events onto a process already loaded with work.
- Be aware that some Cisco IOS primitives are not explicitly associated with the scheduler. If these primitives are called, the scheduler can suspend your process if another process is ready to run.
- If you do not observe recommendations that your process frequently return control of the CPU to the scheduler, your process will be identified by the system as a CPU hog for all to see. Further, if you fail to surrender the CPU for a very long period of time (more than one minute), the Cisco IOS code will assume that the router is hung in an infinite loop and will fire a watchdog timer that will cause the router to reload.

A.3 Using C in the Cisco IOS Source Code

So you think that all there is to writing Cisco IOS code in C is heaving curly braces into an Emacs buffer? (What? You're not using Emacs? Too bad. You must like doing extra work.) Guess again.

A.3.1 Use ANSI C

The current compiler used by Cisco IOS engineering is GCC, the Gnu CC compiler. This compiler has significant features that make it robust for our code development. One of the most notable features of GCC is its ability to enforce ANSI C compliance in the source code by invoking switches, which causes the compiler to issue warning messages when it encounters non-ANSI source code.

The following is a good language reference for ANSI C:

The Annotated ANSI C Standar
American National Standard for Programming Languages—C
Annotated by Herbert Schildt
ANSI/ISO 9899-1990
ISBN 0-07-881952-0

You can also refer to the ANSI C standard itself, but the examples and background information presented by Schildt are very useful.

To find the reference manual for GCC, refer to the GNU documentation at `http://wwwin-swtools.cisco.com/Eng/Release/SWTools/Tools/`. You can also use GNU Emacs Info-mode, but it is unlikely that the information pages will be kept current.

Some coding habits that were commonplace in Kernighan & Ritchie (K&R) C environments are unacceptable in Cisco IOS source code:

- In K&R, functions are not prototyped. In Cisco IOS code, prototype your functions.
- In K&R, `int` is used as a function return type when no valid value is returned or checked, and `void` is used when nothing meaningful is returned. This coding style is not acceptable in Cisco IOS code.

- In K&R, `#define` macros are used when a static inline function would be more readable. In general, there is little reason anymore to use `#define` to replicate code inline.

The GCC compiler is one of the best tools we have to reliably implement the necessary features and speed. Use its features to your best advantage.

A.3.2 Fifty Ways to Shoot Yourself in the Foot

About 10 years ago, a wag in Datamation remarked, "C is a language for consenting adults, Pascal is a language for children, and Ada is a language for hardened criminals."

The writer was referring to how closely the compilers for these languages herd programmers into doing things The Right Way. C allows an engineer to solve the problem at hand with a great deal of freedom and personal discretion. However, just because the compiler *allows* you to do something in a quick-and-dirty fashion does not imply that you *should use* the quick-and-dirty solution. With the freedom from constraints offered by the C language comes the responsibility not to abuse this freedom. When the language offers a more robust and structured solution to the problem at hand, choose that solution even though it might require additional effort.

Although ANSI C offers much more data type checking than K&R compilers do, C is largely promiscuous about what it accepts. C does not check array boundaries, and it does not place checks on your pointer conversions or mathematics to ensure that your result is pointing to something valid. C allows you to overflow simple numeric calculations silently and with stunning efficiency. C allows you to scribble over your stack frame, which will guarantee that when you return from the function you are currently in, you will return to the Land of Oz. In short, C does not hold your hand when you need help, nor will it smack you on the wrist when you err.

As such, the demands placed on the programmer are far greater in C than in some other languages. You alone are responsible for checking your intermediate results for conformance with reality before your code makes assumptions for later execution. It is to your advantage to use what is available in the ANSI C language to help you write code that does what you mean, not just what you say:

Function Prototypes

Use function prototypes. There is no excuse for not using them. You should define a function prototype only once in the entire source, in one header file. Defining multiple prototypes defeats the purpose of having a function prototype in the first place.

Order of Functions within a File

Deliberately arrange the order in which the functions are defined in a file so as to make forward declarations unnecessary and your source easier to maintain and follow. Typically, you do this by placing the function that calls most other static functions at the bottom of the source file and placing the functions at the bottom of the caller/called hierarchy towards the top of the file.

Typecasting

Do not use gratuitous typecasting. One of the most annoying things to observe in C code is a typecast that, with more careful attention to coding, would not be needed. The vast majority of the cases in which typecasting is used are a direct result of either ignorance ("I didn't know that I could do that without casting"), apathy ("So what? It doesn't slow anything down"), or both ("I'm both stupid and slovenly, so there!").

Functions that return `void *` (known as an *opaque type*) need *no* typecasting of the returned pointer. Doing so defeats the whole purpose of declaring the function with an opaque type.

Obscure C Features

Avoid using obscure C language features, such as the "," operator.

Ensuring Correct Results

Do not depend on nonobvious associativity and side effects for correct results. It is unwise to rely on the order of evaluation of side effects performed on variables passed to functions, and it is *extremely* unwise to use side effects in the invocation of a macro.

Static Class

Use the `static` storage class to reduce the scope of variables and functions. Do not make any variable or function an `extern` unless it is used outside the file in which it is defined. This not only helps to keep the name space cleaner, but it also reduces the size of the symbol table passed to the linker. Smaller symbol tables in the link step of the build mean faster link times.

Const Qualifier

Use `const` type qualifiers to allow the compiler to enforce read-only declarations of read-only global storage and pointers that should not change at run time. Without memory protection, there is no way at run-time to prevent someone from writing code that creates a pointer to a non-`const` variable, sets the value of this pointer to point to your storage that has been declared `const`, and subsequently overwrites the storage that had been defined as read only. However, using `const` allows the compiler to flag code that might try to write directly to what you want to be read-only storage.

In addition to declaring initialized storage to be `const`, you should also declare and define the read-only parameters of functions to be `const`.

Register Storage Class

Do not use the `register` storage class unless you are sure—and have verified by looking at the generated code—that using the `register` declaration generates better code than not using it. GCC does not treat the `register` storage class as a mandate, only a "strong hint."

Format of Data Structures

Do not make assumptions about the layout or padding of a structure or the allocated size of a data structure. These depend on the compiler implementation and can vary significantly with the type of target CPU.

Conversion from Signed to Unsigned Types

Pay attention to the conversion from `signed` to `unsigned` types. This is one area where the transition from K&R C to ANSI C occasionally surprises people.

EnumeratedTypes and #defines

When possible, use ordinals declared using `enum`, especially for arguments to `switch` statements. GCC will warn you about unhandled values, a warning you will not receive when using arguments of type `int` or unsigned integers.

If you are creating a list of `#define` macros for enumerated constants, consider whether you can use a real enumerated type rather than the `#define` idiom. You can assign explicit values to enumerated type names if required, as shown in this example:

```
enum {
    first_value=1,
    next_value=2,
    last_value=3
}
```

If you must use the `#define` idiom, write the `#define`s so they are self-indexing. There are few reasons to use the `#define` idiom instead of an enumerated type. One reason would be that the constant definitions can be processed by both the C preprocessor and other, non-C, macro languages. For example:

```
#define EXAMPLE_ITEM_ONE (1)
#define EXAMPLE_ITEM_TWO (EXAMPLE_ITEM_ONE+1)
#define EXAMPLE_ITEM_THREE (EXAMPLE_ITEM_TWO+1)
```

Passing Structures

Do not pass large structures (structures larger than 12 bytes or so) by value to a function, especially on code targeted to RISC architectures. One of the "silent" changes between K&R C and ANSI C is that while K&R C always passes structures by reference to a function, ANSI C makes it possible to pass structures by value. Likewise, do not return structures larger than 4 bytes from a function. Instead, return a pointer to the result.

Mixing C and Assembly Language

GCC allows you to insert assembly language instructions directly into your C code. For reasons of readability and maintenance, we recommend that you contain all such code in as few source files as possible and do not spread such constructs widely across the source code.

Floating-Point Operations

While it might seem outrageous that we even need to mention it, using floating-point operations in Cisco IOS source code is a great way to have someone else shoot you in the foot. Many of our router platforms (the 680*x*0-based platforms) have no floating-point hardware and never will, and the MIPS-based platforms use their floating-point scratch registers for saving interrupt context. If you use floating point math on a MIPS-based platform, you will crash the router in mysterious ways, especially if you are not intimately familiar with the interrupt handler for MIPS-based platforms.

Floating point is unnecessary. Those who offer protestations to the contrary will be flogged. Besides, the only people who like floating-point code are pipe stress freaks and crystallography weenies.

Header Files

Bracket the contents of header (`.h`) files with conditional compilation statements to allow multiple inclusion without generating errors, as shown in this example:

```
#ifndef __FILENAME_H__
#define __FILENAME_H__
/* Contents here. */
#endif
```

A.4 Presentation of the Cisco IOS Source Code

Presentation of the Cisco IOS source code is also known as *pretty printing*. Although the arrangement of white space in the code has little, if any, net effect on code operation, it affects the speed with which your fellow engineers can read and understand your code. Unless you think that unreadable code assures you some measure of job security, there is no logical reason why you would not want your fellow engineers to understand your code as clearly and quickly as possible. (Unreadable code does not lead to job security. Far from it, such code will more than likely lead to your fellow engineers' discontent.) Nonetheless, pretty-printing conventions seem to be one of the issues in software engineering that generate a most rancorous debate.

Fortunately, there are tools that make maintaining a consistent pretty printing and white space very easy. Both the GNU Emacs editor and the `indent` utility allow for easy formatting of C code with a minimum of manual effort.

A.4.1 Specific Code Formatting Issues

The following points address specific code formatting issues:

Standard Cisco Header

Each file should have a standard Cisco header. Templates for the headers of all common types of source files are in the `sys/templates/header.*`.

#include Directives

All `#include` directives should occur after the file header.

Standard Indention

The standard indention is four spaces. The exception is `switch` statements, where the `case` statements are kept at the same indention level as the enclosing `switch {...}`.

Spaces in Function Definitions and Prototypes

In the function definition, there should be a space following a function name and before the opening parenthesis of the argument list. In the prototype argument list in function declarations, there should be no space between the function name and the opening parenthesis. The following is an example of this formatting style:

In the file `boojums.h`:

```
extern void boojum(int, struct snark *);
```

In the file `boojums.c`:

```
void boojum (int arg1, struct snark *ptr)
{
/*
 * Do a bunch of stuff.
 */
}
```

If...Else Statements

The `else` clause of an `if {...} else {...}` should be "cuddled" as shown in this example:

```
if (boojum) {
/* Do some stuff here. */
} else {
/* Do something else. */
}
```

Spaces around Parentheses

Put a space between flow control reserved words and the opening parenthesis of the control statement's test expression:

Correct: `if (test_variable)`
Incorrect: `if(test_variable)`

The `return` statement should follow the same rule:

Correct: `return (TRUE):`
Incorrect: `return(TRUE)`

There should not be a space between the name of a function and the opening parenthesis of the actual argument list:

Correct: `ret_value = boojum_function(arg1, arg2, arg3);`
Incorrect: `ret_value = boojum_function (arg1, arg2, arg3);`

Stubbing Out code

Do not use `#if 0...#endif` to "stub out" code. To stub out code, use an undefined preprocessor variable that gives some clue about why the code is stubbed out, with comments indicating why the code is stubbed out, by whom and when it might be used. For example:

```
/*
* This feature will be enabled in release 10.0(2)
*/
#ifdef TO_BE_ENABLED_IN_RELEASE_100_2
...
#endif
```

An exception to this rule is debugging code, which you place under a conditional compilation variable `DEBUG` or `GLOBAL_DEBUG:`

```
#ifdef DEBUG
...
#endif
```

Formatting Block Comments

Format block comments as follows:

```
/*
* This is a block comment. Don't embellish these with lots of "stars
* and bars."
*/
```

A.4.2 Some Comments about Comments

Yes, we've all heard the refrain "comment your code" until we're all tired to death of hearing it. Well, this is a perfect place to say it again, but with some more specific points.

Comments should tell the reader something non-obvious. A comment that repeats what is blindingly obvious is annoying at best. The following is an example:

```
boojum += 10;    /* Add ten to boojum */
```

This tells you a lot, doesn't it?

It is often better to aggregate comments about high-level and architectural issues in one place, to allow the reader and maintainers of your code to learn much more in a shorter time than if they had to piece together the issues and ideas about your features from comments strewn throughout several files. A good example of aggregating comments is the large block comments in the files `iprouting/igrp2.c` and `iprouting/dual.c`. These comments explain the large issues of the Enhanced IGRP transport protocol and DUAL routing engine, respectively.

Keep comments up to date with the code. Comments that no longer accurately represent what the code is doing are often worse than nonexistent comments.

Devote block comments be devoted to content, not fancy, exquisitely formatted "stars and bars" borders.

If you're about to execute one of those stunningly elegant, minimalist representations of excessive cleverness that C allows all too easily, give the reader a clue about what the outcome of your little pearl of syntax construction should be. (But better than that, don't become yet another obfuscated C coder: rewrite your stunning little pearl.)

A.5 Variable and Storage Persistence, Scope, and Naming

For variables, functions, and program storage, use the minimal scoping required to get the job done. In other words, do not define a variable to be `static` when an `auto` will do the job, and do not define a variable to be `extern` if a `static` scoping will work.

The naming of variables and functions must adhere to different standards according to their scope. `auto` variables must be unique only within the function in which they are declared. `static` variables and functions must be unique within their compilation unit. External variables and functions must be unique throughout the entire lot of the source code going into the link step. As such, prefix variables and functions defined with `extern` scoping with a well-understood and consistent prefix that identifies the module and subsystem, and use these prefixes for all `extern` variables and functions contained within the module or subsystem. The prefix should be at least two characters long.

The following are well-known and obvious prefixes for `extern` variables and functions:

- `ip`
- `atalk`
- `idb`
- `sched`

The prefix `my` is neither well known nor obvious.

A.6 Coding for Reliability

Even if your algorithms and code implement all of a specification or requirements document, your code can and will be subject to incorrect, out-of-specification, or malicious data. In order for your code to survive (and for the router not to crash), you must check for out-of-specification or unexpected data values and act accordingly and reliably when such input values are passed to your code. This is sometimes called *coding defensivel* . This section gives examples of defensive coding.

Checking NULL Pointers

Check for `NULL` pointers passed into your externally visible functions. If you choose to assume that you have passed valid data to your internal or helper functions, that is fine because you are directly responsible for validating the arguments that are passed between your internal code. But for data coming into your modules from other, possibly unknown, areas of the Cisco IOS code, never assume that you have been passed a non-`NULL` pointer, much less a valid one.

Specifying Default Cases

Include default cases on all `switch` statements, and do not allow an unhandled value to fall through into the code that follows the `switch` statement. Also, make note of when you intend a `case` of a `switch` statement to fall through into the next `case` selector. For example:

```
switch (number) {
case 0:
    buginf("You entered zero. I assume you meant 1.\n");
/* Fall through */
case 1: frobozz():
break;
case 2:
...
}
```

Pointer Arithmetic

Do not perform pointer arithmetic based on values computed or received in packets from outside the router without checking the result of the pointer arithmetic for sanity. This is not only a reliability issue, but also sometimes a security issue.

Pointers within Structures

Check pointers contained within structures that point to other structures to ensure that they are non-`NULL` before assuming that they are good pointers.

Checking the malloc() and getbuffer Return

It seems obvious, but it needs to be said: when you call `malloc()` or `getbuffer()`, check to see that the pointer returned was not `NULL`.

Arithmetic Overflow

Check for arithmetic overflow in cases where it would result in a wildly bogus result. There used to be a whole class of especially embarrassing bugs in the router that were caused by arithmetic overflow in timer variables as millisecond timers in the router overflowed from signed to unsigned 32-bit ranges at 24.45 days after boot time.

Data Alignment

Never assume that data values of greater than 8 bits in size are aligned on their natural boundaries in packet or network data. Always use the `GETSHORT` and `GETLONG` macros to read large atomic data in packet buffers, and the `PUTSHORT` and `PUTLONG` macros to write large atomic data.

The following examples of code fragments that are scattered throughout the system—and which you might think are there for reliability—are actually examples of band-aids patched on top of poor designs:

- Using `validmem()` to check pointers contained in your structures because you have designed in a race condition where one thread of execution might be using a pointer to a block of `malloc`'d memory and another thread might be `free`ing the same block. `validmem()` is an expensiv function. Redesign your data structure and all code that uses this function to handle concurrent access.
- Using `onintstack()` to determine when your code is being called from an interrupt. Make every effort to minimize the code executed in interrupts in the router. The more time the router spends in an interrupt code path, the fewer interrupts the router can service, which on most hardware platforms translates directly into a decrease in router throughput. When there are valid reasons to be aware of when your code is executing in an interrupt, call `onintstack()` once and save the result, passing it into the functions that need to know this information.
- `raise_interrupt_level()` and `reset_interrupt_level()` calls around large sections of code indicate that concurrent access to a data structure shared between multiple threads has not been well designed. In this case, the one thread sharing access to the data structures is the interrupt thread. Using this technique to disable interrupts is valid, but only around the small sections of code where the shared data structures are manipulated.
- Writing assertions and `buginf()` messages when an anomaly is detected, and then continuing execution as though nothing were wrong, is a great example of useless code that adds no reliability. If you have checked for a condition that requires the user be informed, then do something smart about the condition.

A.7 Coding for Performance

You should be concerned with three levels of performance in the Cisco IOS code:

- Performance of Algorithms and Data Structures
- Performance Resulting from Use and Abuse of the Cisco IOS Infrastructure
- Instruction-level Performance

Large-scale performance issues are more heavily influenced by the choice of data structures and algorithms, and the wise use of the Cisco IOS fundamental services than by instruction-level optimization. The exceptions to this are well-defined code paths during hardware interrupts, with the most frequent example being the fast-switching implementations for the various protocols.

How do you know you have a performance problem? You use the profiling capability that is built into the Cisco IOS code and run a select set of tests to exercise your code to find the "hot spots." As a rule, unless something is blatantly wasteful of memory and CPU, you should work to get your code executing correctly first before worrying about speeding it up.

In general, it is best to address the performance issues arising from the choice of data structure and algorithm first, before worrying about instruction-level tuning. As an example, it would be pointless to be counting instructions in a protocol's fast-switching code if the data structure for the protocol's fast-switching cache were a linear, singly linked list that contained 3,000 unsorted entries. Any improvements you get from eliminating 10 instructions will be dwarfed by several orders of magnitude of poor performance caused by an ill-chosen algorithm and data structure.

A.7.1 Performance of Algorithms and Data Structures

Reams of paper and millions of trees have been expended publishing books about data structures and algorithms. Many of these books are good, some are great, and some are utter tripe and twaddle. All we will do here is recommend the following and direct you to read them:

- *Introduction to Algorithms*, by Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest, MIT Press, ISBN 0-262-03141-8

  This book is by far one of the most comprehensive data structures and algorithms books available. It is clearly written and gives excellent treatment of the subject of estimating resource usage. In addition, the first six chapters give an excellent review of discrete mathematics and probability. For those looking for a collection of lots of data structures and algorithms in one place, this is a good reference volume. There are errata available from MIT Press by an e-mail responder.

- *The Art of Computer Programming, Fundamental Algorithms*, Second Edition, by Donald Knuth, Addison-Wesley, 1973, ISBN 0-201-03809-9
  The Art of Computer Programming, Sorting and Searching, by Donald Knuth, Addison-Wesley 1973, ISBN 0-201-03803-X

  This is a series of references that should need no introduction.Volumes 1 and 3 are most applicable to Cisco IOS programming. Volume 4 is due to appear in 1997, and the preliminary table of contents indicates that it should be applicable to those working on routing protocols.

The importance of the proper choice of algorithms and data structures in Cisco IOS features and code cannot be underestimated. Cisco routers are used to build some of the largest networks in the world. As such, your features and code should be designed to scale into very large networks. While a hash table might be a suitable method of retrieving items based on a key in a smaller router, a hash table might not be suitable in the networks that run on Cisco IOS software. All the effort spent explaining the "big-Oh" worst-case running time in data structures books is no longer a mathematical abstraction, but a very real difference between having to rewrite large pieces of functionality and providing out-of-the-box customer satisfaction. Several routing protocols in the Cisco IOS software have had to be rewritten because fundamental data structures were chosen improperly when they were first written.

Before implementing a data structure, read the *Cisco IOS Programmer's Guide* and the *Cisco IOS API Reference*. Many data structures that allow the Cisco IOS code to scale into the largest networks in our customer base have already been implemented, tested, and used for many releases of the Cisco IOS software. If you cannot find what you need in existing Cisco IOS code, and if the data structure or algorithm might have use in other places in Cisco IOS code, write the data structure so that it can be used by other parts of the code.

When implementing a complicated data structure, consider that memory in our router products is a finite and expensive resource. Unlike target environments of UNIX, VMS, and other demand-paged virtual memory systems, there is no configuration knob that allows more memory to magically appear.

A.7.2 Performance Resulting from Use and Abuse of the Cisco IOS Infrastructure

You could use the best algorithm, write the slickest code possible, and your feature could still run as fast as a sweating pig stuck in the Georgia mud in August. Why? More than likely, you are misusing the Cisco IOS primitives, and you are not being smart in how you use the following commonly misused Cisco IOS facilities:

- `malloc()` and `free()`—If you find yourself frequently creating and destroying many fixed-sized blocks, consider creating a chunk and using the chunk manager. See the *Cisco IOS Programmer's Guide* for more details. If you find that the chunk manager does not meet your needs, consider a private free list or other ways to avoid calling `malloc()` and `free()` where possible.
- `sprintf()`—Try to do as much formatting in one call as possible, rather than spreading your string formatting out over many small calls to `sprintf()`.
- `raise_interrupt_level()` and `reset_interrupt_level()`—Disabling interrupts to protect against concurrent access to data structures is a poor, but sometimes unavoidable, method. You should disable interrupts for as little time as possible, most specifically around the very small areas that need to be protected against concurrent access. For instance, you do not need to disable interrupts, walk a linked list, unlink or delete the item in question and re-enable interrupts. All that needs to be protected is the actual unlinking operation. A far more preferable primitive to protect access in data structures shared between threads is semaphores.
- `memcmp()` (previously `bcmp()`)—Using these functions to compare Ethernet, Token Ring, and FDDI MAC-level addresses is wasteful. There are macros that perform the required number of word comparisons inline.
- Managed timers—You should generally not use managed timers to implement accounting or simple timestamps. Managed timers are more expensive, both in time and CPU usage, and have excellent applications. However, timestamps are not one of them.

In general, remember that the Cisco IOS software offers a rich set of primitives and services for you to use, but they are not free.

A.7.3 Instruction-level Performance

There are two areas where you can affect instruction-level performance in the Cisco IOS software and on the hardware used in the routers:

- Write code that is easier for the compiler to optimize.
- Write code that is more agreeable to the CPU's memory architecture assumptions.

A.7.3.1 Helping GCC Turn Glop into Gold

When it comes to code generation, many people think that C is the next best thing to a macro assembler on steroids. That might have been the case when we were all writing C on PDP-11s (for those of you too young to remember the PDP-11, let's clear this up right now: the PDP-11 was the single best computer ever created for an assembly programmer), but it is not the case in today's RISC

architectures. Further complicating the issue is the simple fact that the Cisco IOS software contains so much more code in its execution paths than many, if not most, software products written in C, and the Cisco IOS software is by definition and market requirements an embedded, real-time application. A dozen wasted instructions here, a couple of dozen wasted dereferences there, and when you repeat this across millions of lines of code, it starts to add up.

Instruction-level optimization is the wrong thing to do at the beginning of the development cycle. You should "make it right, then make it fast," and concern yourself more with algorithms and data structure optimization in the early part of the development cycle. Few things in the Cisco IOS source require instruction-level optimization beyond what the compiler will do.

To aid you in getting the most out of GCC's optimization, consider these issues when writing code:

- Inline functions. The best candidates for inlining are functions that are small and limit their side effects, and for which the call/return overhead would comprise a significant portion of their total execution time. Be careful, however, because inlining functions with wild abandon can quickly bloat the size of the resulting image.

  In general, define functions as `inline` only when they are small and well defined, and there is a compelling reason of speed to expand the functions inline.

- Repeated code. Hoist code that is repeated in both the `if` and `else` clauses out of the conditional, either above the conditional or below it, as appropriate. Certainly, GCC can do this, but if you do it, you know it is done.

- Register declarations. Do not declare variables with the register allocation unless you have examined the resulting code and determined that the generated code is actually better with the register declaration than without it. GCC does not consider a register declaration to be imperative. Rather, GCC uses `register` declarations as a "strong hint" in the optimization phase. Also note that when you use a `register` declaration, you cannot take the address of the variable so declared.

- `volatile` keyword. When you do not want a variable promoted to a register during optimization, use the `volatile` keyword. You typically do this for variables that are changed by hardware interrupts behind your back. If you do not declare things like memory-mapped hardware-manipulated locations to be `volatile`, you can get some very surprising execution results.

  The `volatile` keyword also affects instruction re-ordering, particularly on RISC architectures. If instruction order is important to the intermediate values in an expression, the variables must be declared `volatile`.

- Multiple dereferencing. If you find that you are performing the same double-dereference (or triple-dereference) more than a very few times in the same function, it is likely to be more effective to cache the result of the first dereference in a local variable. For example, if the following type of pattern occurs several times in a function:

  ```
  if (idb->hwptr->status & IDB_ETHER) {...
  ```

  GCC reloads a register with `struct1->struct2` every time you write the above expression. In this case, the most effective way to code is to cache the final 32-bit value of the expression and use it where necessary. Even if you are dereferencing `idb->hwptr` several times to access different fields of `struct2`, it is more efficient to cache the value of `struct2`.

- This is rather esoteric, but for those portions of your code that truly must be as fast as possible, pay attention to whether your target CPU prefers to do branch prediction for the branches taken or not taken, and write your code according. Should you choose to do branch prediction, examine the output of the compiler to verify that you get the instruction stream you expect, and comment the code to indicate why the speed of the code depends on such innocuous things as the sense of the conditional tests.

- Bit field instructions are not terribly fast. Use them where necessary, but keep in mind that some of our target CPUs do not have bit field instructions. This is particularly true of the QUICC chip, which has an ALU that is basically a 68020 instruction set without bit field and rotate instructions.
- If you are copying most or all fields of a structure from one instance of the structure to another instance, consider using either `memcpy()` or a `struct` copy. A `struct` copy of a small structure (32 bytes or less on 680*x*0 systems and 64 bytes or less on RISC systems) generates a sequence of inline instructions to copy the fields of the structure. For larger structures, a call to `memcpy()` or `bcopy()` is generated by the compiler.
- Bit fields in C structures are one of the few features of the language that are a speed trap. If you are using single bit values in a structure bit field for true/false values, you can realize much better speed with an array of `unsigned char`, especially on RISC architectures. Further, you should never use C bit fields in data structures that are passed across the network. A C compiler is, by definition, free to implement bit fields in a structure any way it sees fit. The bits can be allocated starting at whichever end of the machine word the compiler chooses, with any padding necessary to the target architecture. So while the C definition of a data structure passed on the network might read as exactly the same on two different implementations, the code might produce very different results on the wire.

  The advantage of using bit fields in C structures is that it is semantically cleaner and easier to understand the author's intent, especially if more bits are defined in aggregate than fit in one machine word.

A.7.3.2 Not All Memories Are Golden

The speed with which the CPU accesses the memory in your data structures can vary widely if you do not pay attention to how your data is aligned and accessed. Follow these guidelines to maximize CPU access speed:

- Align your data, especially structure elements, on their native boundaries. This means that `shorts` should start on any even address, `longwords` on addresses evenly divisible by four, and 64-bit values (`quadwords`) on addresses evenly divisible by eight. On some CPUs, such as the 680*x*0, failure to properly align your data results in a significant performance impact because the misalignment is handled by the hardware. On other CPU families, such as the MIPS architectures, the performance impact is even greater because the exception is handled in low-level software.
- Use "natural" sizes. If a target architecture accesses a 16-bit memory location faster than it accesses an 8-bit location, consider declaring your storage to be a 16-bit large area. You must then weigh this increase in size against other factors, such as cache hit ratios and cache line packing.
- If several fields of a structure are accessed during speed-critical code paths but many more that are not, group all the fields of the structure that are accessed in the speed-critical code path(s) together in the `struct`, even if doing so might not be as aesthetically pleasing as you would like. By grouping the fields together, you increase the likelihood that more of them will be pulled into cache memory on the first access, and you reduce the number of cache entries required to hold all your speed-critical fields.
- Do not misuse special memory regions. In some router architectures, there are small regions of specialized memory, such as shared, nonvolatile, and Flash memory. If you need to read something from these memory regions, do it once and cache the results in normal processor memory. Your code will be much faster for having done so.

APPENDIX B

Cisco IOS Software Organization

This appendix lists many of the subsystems in Cisco IOS Releases 11.1 and 11.2. The purpose of this appendix is to give you an idea of how the Cisco IOS software is organized. The information presented here is based on the subsystems available for use by a Cisco 2500 platform. Other platforms might contain additional or fewer subsystems, and will contain platform-specific code.

For information about the contents of Cisco IOS images, see `http://wwwin-eng/Eng/Release/subsettool/audit`.

B.1 Description of the Cisco IOS Subsystems

Table B-1 through Table B - 13 list many of the subsystems in Cisco IOS Releases 11.1 and 11.2.

Table B-1 Cisco IOS LAN Protocol Subsystems

Subsystem	Description
`shr_atmib2.o`	Address translation MIB
`shr_arap.o`	AppleTalk Remote Access (ARA) protocol
`shr_arp.o`	Address Resolution Protocol (AARP)

`shr_atalk.o`
`shr_atalkmib.o`
`shr_atalktest.o`
`shr_atfast_les.o`
`ataurp.o` (Release 11.2 only)
`atcp.o` (Release 11.2 only)
	AppleTalk, AppleTalk fast switching, and AURP

`shr_smrp.o`
`shr_smrptest.o`
`shr_atsmrp.o`
`at_smrpfast.o`
`at_smrpfast_les.o`
	AppleTalk Simple Multicast Routing Protocol (SMRP)

`shr_atdomain.o`	AppleTalk domain support
`shr_ateigrp.o`	AppleTalk Enhanced IGRP
`shr_atip.o`	AppleTalk IP support
`tfast_les.o`	Fast tunnel for low-end systems
`shr_tunnel.o`	Virtual interface that acts like a point-to-point link over IP

Subsystem	Description
vinesfast_les.o shr_vines.o shr_vinesmib.o shr_vinestest.o	Banyan VINES protocol support
shr_tarp.o	Target ID for the Address Resolution Protocol (ARP)
shr_bgpmib.o shr_bgp.o	Border Gateway Protocol (BGP) and MIB
shr_chat.o	Chat script processing
shr_cls.o	Cisco link services
ccp.o (Release 11.2 only)	Compression Control Program
cpp.o (Release 11.2 only)	Combinet Packet Protocol
dnfast_les.o shr_decnet.o shr_decnetmib.o	DECnet
shr_dncnv.o	DECnet Phase 4-to-Phase 5 conversion
dhcp_client.o (shr_dhcp.o in Releas e11.1)	Dynamic Host Configuration Protocol
shr_egp.o shr_egpmib2.o	Exterior Gateway Protocol (EGP)
shr_eigrp.o	Enhanced IGRP
shr_routing.o	Integrated routing subsystem
shr_mop.o	Maintenance Operation Protocol (MOP) booting for DEC environments
shr_ftp.o	File Transfer Protocol (FTP)
shr_gre.o	Generic Route Encapsulation (GRE)
shr_icmpmib2.o	Internet Control Message Protocol (ICMP) MIB
shr_ident.o	RFC1413 Ident protocol
shr_igrp.o	Interior Gateway Routing Protocol (IGRP)
sh_ip_policy.o	IP policy routing
shr_ipip.o	Raw IP-over-IP support
ipasm.o ipfast.o ipfast_les.o	IP fast switching
shr_iprouting.o	Generic IP routing functions
shr_ipservices.o shr_tcpmib2.o	Basic IP services, including the TCP driver and DNSIX
ipttcp.o	TTCP command, which is used to generate TCP traffic from one router to another (or one router to a workstation) to measure network and protocol stack performance
http.o	HTTP server
shr_rip.o	Routing Information Protocol (RIP) for IP

Subsystem	Description
shr_hpprobe.o (Release 11.2 only) shr_ipaccount.o (Release 11.2 only) shr_ipalias.o (Release 11.2 only) shr_ipboot.o (Release 11.2 only) shr_ipbootp.o (Release 11.2 only) shr_ipcdp.o (Release 11.2 only) shr_ipcompress.o (Release 11.2 only) shr_ipcore.o (Release 11.2 only) shr_ipdiag.o (Release 11.2 only) shr_ipdns.o (Release 11.2 only) shr_ipgdp.o (Release 11.2 only) shr_iprarp.o (Release 11.2 only) shr_ipudptcp.o (Release 11.2 only) shr_tacacs.o (Release 11.2 only) shr_udpmib2.o (Release 11.2 only) shr_iphost.o (Release 11.1 only)	IP host functions
udp_flood_fs.0	UDP fast-switch flooding
ipnacl.o (Release 11.2 only)	IP named address lists
shr_ipmib2.o shr_ipmroutemib.o shr_igmpmib.o shr_ipmulticast.o ipmfast_les.o	IP multicast
rsvp.o rsvpmib.o	Resource Reservation Protocol and MIB
traffic_shape.o	Traffic shaping routines
ipnat.o (Release 11.2 only)	IP network address translation
shr_pimmib.o	Protocol Independent Multicast (PIM)
shr_ripsapmib.o shr_novellmib.o shr_ipxmib.o novfast_les.o shr_ipx.o ipxnasi.o ipxwan.o (Release 11.2 only) ipxeigrp.o (Release 11.2 only) ipxcp.o (Release 11.2 only)	Novell IPX
shr_ipxcompression.o	IPX compression
shr_ipxnhrp.o	IPX Next Hop Routing Protocol (NHRP)
shr_ipxnlsp.o shr_nlspmib.o	IPX NetWare Link Services Protocol (NLSP) and MIB
shr_isis.o shr_isis_clns.o shr_isis_ip.o shr_isis_nlsp_debug.o	Intermediate System-to-Intermediate System (IS-IS) dynamic routing protocol
shr_eon.o	EON-compatible ISO CLNS-over-IP tunneling
clnsfast_les.o shr_clns.o shr_clns_adj.o	ISO Connectionless Network Protocol (CLNS)
shr_lpd.o	Subset of the UNIX line printer daemon (LPD) protocol

Subsystem	Description
shr_nrhp.o shr_ipnhrp.o	Next Hop Routing Protocol (NHRP)
ntp_refclock.o (Release 11.2 only) ntp_refclock_master.o (Release 11.2 only) ntp_refclock_pps.o (Release 11.2 only) ntp_refclock_telsol.o (Release 11.2 only) shr_ntp.o (Release 11.1 only)	Network Time Protocol (NTP)
shr_ospf.o shr_ospfmib.o	Open Systems Path First (OSPF) routing protocol and MIB
xnsfast_les.o shr_xns.o shr_xnsmib.o	XNS protocol
shr_griproute.o	Routing Information Protocol (RIP) for XNS, Apollo Domain, and Ungermann-Bass
shr_apollo.o	Apollo Domain

Table B-2 Cisco IOS WAN Protocol Subsystems

Subsystem	Description
shr_atm_dxi.o	ATM DXI
sr_atommib_es.o (Release 11.2 only)	AToM MIB support for end stations
shr_compress.o	Generic compression
sub_callmib.o shr_dialer.o	Dialer support for dialers attached to serial interfaces; used for DDR, BOD, and ISDN
shr_frmib.o fr_fast_les.o shr_frame.o frame_arp.o (Release 11.2 only) frame_svc.o (Release 11.2 only) frame_traffic.o (Release 11.2 only) frame_tunnel.o (Release 11.2 only)	Frame Relay
sub_isdn.o sub_isdnmib.o	ISDN and MIB
shr_ppp.o ipcp.o	Point-to-Point Protocol (PPP)
mlpvt.o (Release 11.2 only)	Multichassis multilink PPP
shr_smds.o	SMDS
shr_snapshot.o shr_snapshotmib.o	Snapshot routing
shr_v120.o	V.120 protocol over ISDN
shr_lapbmib.o shr_pad.o shr_x25.o shr_x25mib.o	X.25

Table B-3 Cisco IOS Bridging Subsystems

Subsystem	Description
shr_rsrbmib.o shr_bridge_rsrb_ui.o	Remote source-route bridging (RSRB)
fastsrb_les.o shr_srbmib.o srbmib.o (Release 11.2 only) shr_bridge_sr.o shr_bridge_srb_ui.o srbcore.o (Release 11.2 only)	Source-route bridging (SRB
tshr_bridge_t.o shr_bridge_t_ui.o tbridge.o tbridge_les.o shr_tbmib.o bridge_t_cmf.o (Release 11.2 only)	Transparent bridging and MIB
vpn.o (Release 11.2 only)	Virtual private dial-up network
shr_vtemplate.o	Virtual template interface services

Table B-4 Cisco IOS Communications Server Subsystems

Subsystem	Description
shr_comm.o	Communications server
shr_config_history.o	Configuration history database
shr_confmanmib.o	Configuration management MIB
crypto.o (Release 11.2 only) cryptomib.o (Release 11.2 only)	Network-layer 56-bit DES encryption and MIB
exportable_crypto.o (Release 11.2 only)	Network-layer 40-bit DES encryption
shr_kerberos.o	Kerberos
keyman.o	Lock and ke
shr_lat.o	Local-area transport (LAT)
shr_menus.o	User-definable menus for accessing Cisco IOS commands
shr_modem_discovery.o	Automatic recognition of modems
shr_modemcap.o	Modem capabilities database
shr_pt.o shr_pt_auto.o shr_pt_lat.o shr_pt_latauto.o shr_pt_latpad.o shr_pt_latslip.o shr_pt_pad.o shr_pt_padauto.o shr_pt_padslip.o shr_pt_padtcp.o shr_pt_slip_ppp.o shr_pt_tcp.o shr_pt_tcpauto.o shr_pt_tcplat.o shr_pt_tcpslip.o	Protocol translation

Subsystem	Description
shr_radius.o	Livingston's "Radius" network authentication protocol
shr_tacacs_plus.o	TACACS+
shr_tn3270.o tn3270s.o (Release 11.2 only) shr_tsmib.o (Release 11.2 only)	TN3270 and MIB
shr_xremote.o	XRemote

Table B-5 Cisco IOS Utilities Subsystems

Subsystem	Description
shr_ifmib.o	Interfaces MI
shr_des.o	Data Encryption Standard (DES)

Table B-6 Cisco IOS Driver Subsystems

Subsystem	Description
sub_pcbus.lnm.o	Network Management for AccessPro ISA
sub_pcbus.o	AccessPro ISA bus driver
sub_c3000.o	Cisco 2500 series-specific support
sub_cd2430.o	Asynchronous driver for the Cisco 2509, Cisco 2510, Cisco 2511, and Cisco 2 5 12 access servers
sub_hub.o	Driver for Cisco 2500 series hubs
sub_brut.o	Console/auxiliary port driver for Cisco 2500 series
shr_ether.o shr_ethermib.o	Generic Ethernet driver
sub_lance.o	Platform-specific Ethernet driver
shr_fastswitch.o	Generic fast switching
shr_flash_les_mib.o	Low-end Flash MIB
shr_flashmib.o	Flash MIB
sub_bri.o	ISDN BRI driver
shr_lex.o	LAN Extender driver
sub_lex_platform.o	LAN Extender platform support
lex_ncp.o (Release 11.2 only)	LAN Extender network control program
shr_serial.o	General serial driver
sub_les_serial.o	Low-end serial driver
sub_hd64570.o	Platform-specific serial driver
sub_rptrmib.o	Repeater MIB
shr_tring.o shr_trmib.o	Generic Token Ring driver
sub_tms380.o	Token Ring platform-specific support
sub_partner.o	Miscellaneous drivers for OEM platforms

Table B-7 Cisco IOS Network Management Subsystems

Subsystem	Description
shr_cdp.o shr_cdpmib.o cdp_ncp.o (Release 11.2 only)	Cisco Discovery Protocol (CDP)
shr_chassismib.o	Physical representation of the platform
entity.o (Release 11.2 only)	Physical and logical managed entities
shr_queuemib.o	Queue MIB
shr_rmon.o shr_rmonlite.o	Remote monitoring
sr_rs232mib.o sr_mempoolmib.o	SNMPv2 bilingual agent code
shr_syslog_history.o shr_syslogmib.o	Syslog messages

Table B-8 Cisco IOS VLAN Subsystems

Subsystem	Description
vlan_les.o ieee_vlan.o isl_vlan.o shr_vlan.o	Virtual LAN

Table B-9 Cisco IOS Kernel Subsystems

Subsystem	Description
shr_core.o (Release 11.2 only) sub_core_platform.o (Release 11.2 only) shr_ukernel.o (Release 11.2 only) sub_ukernel_platform.o (Release 11.2 only) sub_kernel.o (Release 11.1 only)	Cisco IOS kernel. For Release 11.2, the kernel is divided in platform-independent and platform-dependent portions. shr_ukernel.o contains the essentials necessary for the scheduler to operate, including memory management, list support, timer support, subsystem and registry support, and basic clock support. shr_core.o contains other things that were in sub_kernel.o, including packet buffer support, authentication, common access list support, async/TTY interface/modem support, login, compression, host name support, error logging, the printf routine, priority queueing, and virtual interfaces.

Table B-10 Cisco IOS IBM Subsystems

Subsystem	Description
shr_dlur.o shr_appn.o shr_appnmib.o shr_appnutil.o	Advanced Peer-to-Peer Networking (APPN)
sub_bsc.o	Bisync
shr_bstun.o (Release 11.2 only) shr_bstunmib.o (Release 11.2 only) sub_bstun.o (Release 11.1 only) sub_bstunmib.o (Release 11.1 only)	Bisync serial tunnel

Subsystem	Description
shr_dlcsw.o shr_dlc_base.o shr_vdlc.o (Release 11.2 only)	CLSI
fastdlsw_les.o shr_dlsw.o sr_cdlswmib.o	Data Link Switching (DLSw) and MIB
shr_dspu_ui.o shr_dspumib.o	Downstream PU (DSPU)
shr_ibmnm.o	IBM Network Management protocol
shr_lack.o	Local Acknowledgment
shr_lanmgr.o shr_lanmg_ui.o	Token Ring LAN manager
shr_llc2.o	LLC2
ncia.o (Release 11.2 only) ncia_ui.o (Release 11.2 only)	Native Client Interface Architecture (NCIA)
shr_netbios.o netbios_as.o shr_netbios_ui.o	NetBIOS over LLC2 and Novell IPX
shr_netbios_acl.o shr_netbios_acl_i.o ibuint.o (Release 11.2 only)	NetBIOS over LLC2 and Novell IPX access lists
shr_qllc.o	Qualified Logical Link Control (QLLC)
rtt_dspu.o (Release 11.2 only) rtt_mon.o (Release 11.2 only) rtt_monmib.o (Release 11.2 only) rtt_snanm.o (Release 11.2 only)	Response time reporter
shr_sdlc.o shr_sdlc.ui.o	Serial Data Link Control (SDLC)
shr_sdllc.o shr_sdllcmib.o	SDLC media translation
shr_sna.o shr_sna_pu.o	SNA
shr_snanm.o	SNA network management
shr_snasdlcmib.o	SNA SDLC MIB
shr_stunmib.o shr_stun.o shr_stun_ui.o	Serial Tunnel (STUN)

Table B-11 Cisco IOS Library Utility Subsystems

Subsystem	Description
access_expr.o	Boolean expression analyzer
avl.o	AVL trees
fsm.o	General-purpose finite state machine driver
inet_aton.o	Convert an Internet address to a binary address
iso_chksum.o	ISO checksum routines

Subsystem	Description
itemlist.o	Placeholder for files to be added in Release 11.1
libgcc_math.o	64-bit math support culled from GCC libgcc2
md5.o	MD5 source code from RFC 1321
md5_crypt.o	One-way cipher function based on MD5
memmove.o	ANSI/POSIX-compatible memory move routine
msg_radix.o	Message file for radix trees
msg_util.o	Message definitions from utility routines
nsap.o	NSAP address definitions
nsap_filter.o	Filter facility used by CLNSand ATM
peer_util.o (Release 11.1 only)	Common utility routines for DLSw, RSRB, and STUN peers
qsort.o	Quicksort routines
radix.o	Radix tree manipulation package
random.o random_fill.o	Random number generator routines
range.o	Routines for range arithmetic
regexp.o regexp_access.o regsub.o	Routines for regular expressions
rif_util.o (Release 11.1 only)	RIF utilities
sna_util.o (Release 11.1 only)	SNA PIU manipulation utilities
sorted_array.o	Manipulation of sorted arrays
string.o (Release 11.1 only)	Platform-independent standard C library
tree.o	Red-Black trees
wavl.o	Wrapper functions for multithreaded AVL trees

Table B-12 Cisco IOS ANSI Library Subsystems (Release 11.2 only)

Subsystem	Description
_tolower.o	Translate uppercase characters into lowercase
_touuper.o	Translate lowercase characters into uppercase
abs.o	Integer absolute value
atoi.o	ASCII-to-integer conversion routine
atol.o	ASCII-to-integer conversion routine
calloc.o	Allocate and zero memory
div.o	Divide two integers
errno.o	Provide errno
isalnum.o	Determine whether argument is an alphanumeric character
isalpha.o	Determine whether argument is an alphabetic character
isascii.o	Determine whether character is in an ASCII range
iscntrl.o	Determine whether argument is a control character
isdigit.o	Determine whether argument is an ASCII decimal digit

Subsystem	Description
isgraph.o	Determine whether argument is a printable character (except space)
islower.o	Determine whether argument is a lowercase ASCII character
isprint.o	Determine whether argument is a printable character
ispunct.o	Determine whether argument is a punctuation character
isspace.o	Determine whether the argument is a white space character
isupper.o	Determine whether argument is an uppercase character
isxdigit.o	Determine whether argument is a hexadecimal digit
labs.o	Integer absolute value
ldiv.o	Divide two long integers
memchr.o	Scan memory for a byte
memcmp.o	Memory comparison routine
memcpy.o	Copy nonoverlapping memory areas
memset.o	Set the value of a block of memory
reent_init.o	Initialize a reentrancy block
strcat.o	Concatenate two nonoverlapping strings
strchr.o	Search for a character in a string
strcmp.o	Compare two strings
strcoll.o	Compare two strings using the current locale
strcpy.o	Copy a string
strcspn.o	Search a string for characters that are not in the second string
strerror.o	Convert error number to a string
strlen.o	Return string length
strncat.o	Copy a counted nonoverlapping string
strncmp.o	Character string comparison routine
strncpy.o	Counted string copy
strpbrk.o	Find characters in a string
strrchr.o	Reverse search for characters in a string
strstr.o	Find string segment
strtol.o	Convert a number string to a long
strtoul.o	Convert an unsigned number string to an unsigned long
toascii.o	Force integers into ASCII range
tolower.o	Translate uppercase characters into lowercase
toupper.o	Translate lowercase characters into uppercase
wctomb.o	Convert a wide character string to a multibyte character string

Table B-13 Cisco IOS Cisco Library Subsystems (Release 11.2 only)

Subsystem	Description
atohex.o	Convert two ASCII characters into a hexadecimal byte
atoo.o	Convert an ASCII value to octal

Subsystem	Description
badbdc.o	Return nonzero if BCD string has a bad entry
bcdtochar.o	Convert BCD string to a character
bcmp_generic.o	Byte comparison routine
bzero.o	Zero a block of memory
chartohex.o	Convert a character to a hexadecimal nibble
cmpid.o	Compare two byte strings for a specified number of bytes
concat.o	Concatenate two strings to create a third string
deblank.o	Remove leading white space
firstbit.o	Return the bit number of the first bit set from left to right
firstrbit.o	Return the bit number of the first bit set from right to left
ls_string.o	Determine whether string is an ASCII string
lcmp.o	Long compare routine
lowercase.o	Convert a string to lowercase
null.o	Check for NULL or empty string
num_bits.o	Return the number of bits set in an integer
sstrncat.o	Cisco safe version of strncat
sstrncpy.o	Cisco safe version of strncpy
strcasecmp.o	Case-insensitive character string comparison
string_getnext.o	Get a buffer into which to write a short string
strncasecmp.o	Case-insensitive character string comparison
sys_ebcdic_to_ascii.o	Convert from EBCDIC to ASCII
termchar.o	Determine whether character is a space
tohexchar.o	Location for a table of hexadecimal digits
uppercase.o	Convert a string to uppercase

B.2 Description of the IP Subsystems

This section details the object files in the following IP routing protocol subsystems f or CiscoIOS Release 11.1.

- IP Host Subsystem
- IP Routing Subsystem
- IP Services Subsystem

B.2.1 IP Host Subsystem

The IP host subsystem contains object files for the following IP functions:

- Traceroute
- ARP and reverse ARP
- HProbe

- BOOTP
- GDP/IRDP
- TFTP
- TACACS
- TCP core
- TCP compression
- IP support
- Accounting
- Access lists
- rcp
- rsh
- Telnet
- SNMP
- ICMP
- System error logging
- Domain service support

Table B-14 IP Host Subsystem Object Files

Object File	Description

`ipaddress.o`
`ip_init.o`
`ipsupport.o`
`ip_debug.o`
`ip_setup.o`
`ip_test.o`
`msg_ip.o`
`ipname.o`
	Basic nonrouting core IP services

`syslog.o`	Message transmission to syslog daemon
`domain.o`	Domain service support
`ip_media.o`	Media-dependent IP routines
`ipinput.o`	IP input and gateway processing

`ipparse.o`
`ip_actions.o`
`ip_chain.o`
	IP command-line parsing

`ipoptions.o`
`msg_ipsecure.o`
`ipoptparse.o`
	IP security options

`icmp.o` `icmpping.o`	Internet Control Message Protocol (ICMP)
`path.o`	IP routing using ICMP redirects
`iptrace.o`	Traceroute

Object File	Description
ip_arp.o rarp.o	ARP and Reverse ARP
probe.o probe_chain.o	HP Probe (Hewlett-Packard's version of ARP)
bootp.o	BOOTP boot code
gdpclient.o gdpclient_chain.o gdp.o gdp_chain.o	Gateway Discovery Protocol (GDP)
tftp.o tftp_server.o tftp_chain.o tftp_debug.o tftp_util.o	Trivial File Transfer Protocol (TFTP)
tacacs.o xtacacs.o tacacs_chain.o msg_tacacs.o	TACACS
ipaccess1.o ipaccount.o ipaccount_chain.o	IP accounting
tcpfast.o tcpinput.o tcpoutput.o tcpservice.o tcpsupport.o tcptop.o ttcp.o tcpvty.o tcp_chain.o tcp_debug.o msg_tcp.o tuba.o ip_tuba.o tuba_default.o udp.o udp_debug.o	TCP core services
tcpcompress.o tcpcompress_chain.o	TCP compression
rcp.o rsh.o msg_rcmd.o	Remote copy (rcp) and remote shell (rsh)
telnet.o telnet_debug.o msg_telnet.o	Telnet
ipaccess2.o ipaccess_chain.o	IP access lists
ipalias.o ipalias_chain.o	IP aliases
ip_snmp.o msg_snmp.o	SNMP support

B.2.2 IP Routing Subsystem

The IP routing subsystem contains object files for the following IP functions:

- Common IP routing routines
- Next Hop Routing Protocol (NHRP)
- IGRP
- Radix trees
- ICMP router discovery
- Hot standby
- IP community list
- IP mobility

Table B-15 Cisco IOS IP Routing Subsystem Object Files

Object File	Description
`iprouting_init.o`	IP routing protocol initialization

`route1.o`
`route2.o`
`route3.o`
`ipstatic.o`
`ipfast.o`
`ipfast_chain.o`
`route_map.o`
`iprouting_chain.o`
`iprouting_setup.o`
`iprouting_debug.o`
`msg_iproute.o`
`msg_ipfast.o`
`iprouting_actions.o`
`ip_routing.o`
	Common IP routing routines

`ipigrp2.o`	IP IGRP

`radix.o`
`ipradix.o`
`msg_radix.o`
	Radix trees

`irdp.o` `irdp_chain.o`	ICMP router discovery protocol
`community.o`	IP community list-related functions
`standby.o` `msg_standby.o`	Host Standby Routing Protocol (HSRP)
`ipmobile.o` `ipmobile_chain.o`	IP host mobility protocol

`nhrp.o`
`nhrp_cache.o`
`nhrp_vc.o`
`msg_nhrp.o`
`ipnhrp.o`
	Next Hop Routing Protocol (NHRP)

B.2.3 IP Services Subsystem

The IP services subsystem contains object files for the following IP functions:

- TCP driver
- DNSIX

Table B-16 Cisco IOS IP Services Subsystem Object Files

Object File	Description
`tcpdriver.o`	TCP driver

`dmdp.o`
`dnsix_nat.o`
`dnsix_chain.o`
`dnsix_debug.o`
	DNSIX

B.3 Description of the Cisco IOS Kernel Subsystems

This section details the object files in the subsystems for the following Cisco IOS kernel components for Cisco IOS Release 11.1:

- Scheduler Subsystem
- Chain Subsystem
- Media Subsystem
- Parser Subsystem
- Core TTY Subsystem
- Core Router Subsystem
- Core Memory Management, Logging, and Print Subsystem
- Core Time Services and Timer Subsystem
- Core Modular Subsystem
- Miscellaneous Subsystems

B.3.1 Scheduler Subsystem

Table B-17 lists the object files in the Cisco IOS scheduler subsystem.

Table B-17 Scheduler Subsystem Object Files

Object File	Description

`sched.o`
`sched_compatibility.o`
`sched_test.o`
	Scheduler

B.3.2 Chain Subsystem

Table B-18 lists the object files in the Cisco IOS chain subsystem, which contains parse chains and code for Cisco IOS diagnostic functions.

Table B-18 Chain Subsystem Object Files

Object File	Description
`free_chain.o`	Parse chains and code for memory pool commands
`buffers_chain.o`	Parse chains and code for buffer pool commands
`registry_chain`	Parse chains and code for registry commands (in registries.o)
`region_chain.o`	Parse chains and code for memory region commands
`sched_chain.o`	Parse chains and code for scheduler commands
`list_chain.o`	Parse chains and code for list manager support
`subsys_chain.o`	Parse chains and code for subsystem support

B.3.3 Media Subsystem

Table B-19 lists the object files in the media subsystem.

Table B-19 Media Subsystem Object Files

Object File	Description
`ieee.o`	IEEE 802.*x* definitions
`msg_datalink.o`	Message file for generic datalink facility
`static_map.o` `static_mapchain.o`	Support for generic static maps

B.3.4 Parser Subsystem

Table B-20 lists the object files in the parser and EXEC subsystem.

Table B-20 Parser Subsystem Object Files

Object File	Description
`chain.o`	C file into which the token macros, action routines, and token chains are built

`command1.o`
`command2.o`
`command_chain.o`
	EXEC command support

`config.o`	Configuration commands
`parser_util.o`	Utility functions for the parser
`parser.o`	Parser-specific routines
`parser_debug.o`	Debugging routines for the parser
`actions.o`	Action functions for parse tokens
`ctype.o`	Character types

Object File	Description
latgroup.o setup.o	LAT group code handling
exec.o exec_chain.o	EXEC functions
debug.o	Debug command support
msg_parser.o	Parser error messages
alias.o	Command alias functions
mode.o	Parser mode functions
privilege.o	Parser privilege functions

B.3.5 Core TTY Subsystem

Table B-21 lists the object files in the Cisco IOS core TTY subsystem.

Table B-21 Core TTY Subsystem Object Files

Object File	Description
aaa.o aaa_acct.o aaa_chain.o keyman.o	System authentication, authorization, and accounting functions
connect.o connect_chain.o	Connection management services
hostname.o	Host command support
async.o async_chain.o async_debug.o	Asynchronous port support
login.o login_chain.o	Old system authentication (replaced by aaa object files)
modemsupport.o	Modem support
monitor1.o	ROM monitor support
ttycon.o ttysrv.o ttystatem.o	Terminal services

B.3.6 Core Router Subsystem

Table B-22 lists the object files in the core router subsystem.

Table B-22 Core Router Subsystem Object Files

Object File	Description
access.o access_chain.o	Common access list support
if_groups.o	Interface groups
if_vidb.o	Virtual IDB support

Object File	Description
interface.o interface_api.o	Functions for manipulating software and hardware IDB structures
linkdown_event.o	Ethernet, Token Ring, and HDLC link-down handling for SN network management
loopback.o	Support for a virtual interface that acts like a loopback interface
msg_clear.o	Message file for **clear** commands
msg_lineproto.o	Message file for line protocol commands
network.o network_debug.o	Generic network support, including keepalives, hold queue management, interface commands and IDB commands
pak_api.o	Packet interface API
priority.o priority_chain.o	Priority queueing
trace.o	Support for **trace** command
compress_lzw.o config_compress.o	Compression support

B.3.7 Core Memory Management, Logging, and Print Subsystem

Table B-23 lists the object files in the core memory management, logging, and print subsystem.

Table B-23 Core Memory Management, Logging, and Print Subsystem Object Files

Object File	Description
buffers.o chunk.o	Buffer management support
element.o free.o	Memory management support
logger.o logger_chain.o	System logging support
print.o printf.o	System print services
region.o	Region management services

B.3.8 Core Time Services and Timer Subsystem

Table B-24 lists the object files in the core time services and timer subsystem.

Table B-24 Core Time Services and Timer Subsystem Object Files

Object File	Description
clock.o clock_guts.o clock_util.o	Basic system clock and tim support routines

Object File	Description
mgd_timers.o time_utils.o timer.o timer_chain.o	Timer support

B.3.9 Core Modular Subsystem

Table B-25 lists the object files in the core modular subsystem.

Table B-25 Core Modular Subsystem Object Files

Object File	Description
msg_subsys.o registry.o registry_debug.o subsys.o	Registry and subsystem support

B.3.10 Miscellaneous Subsystems

Table B-26 lists the object files in miscellaneous core subsystems.

Table B-26 Miscellaneous Core Subsystem Object Files

Object File	Description
address.o	Functions for manipulating addrtype and hwaddrtype address objects
asm.o	Generic assembler support
boot.o coverage_analyze.o	Network configuration and loading support
delay_table.o	Definitions for the calibration delay loops
init.o	CPU-independent system initialization functions
list.o	List management support
msg_system.o	System error messages
name.o	Protocol-independent host name and address lookup
nv_common.o	System-independent support for nonvolatile configuration memory (NVRAM)
os_debug.o	General Cisco IOS debugging routines
parse_util.o	Useful parse tables such and those for protocol addresses and source files
platform.o	Platform-specific interrupt variables
process.o	Scheduler primitives for process manipulation
profile.o	CPU profiling support
queue.o	Singly linked list support
reload.o	Scheduled reload and message-printing support

Object File	Description
service.o	Support for various services such as finger, line number, PAD, and Telnet
signal.o	Per-thread signal support
sr_core.o	SNMPv1 and SNMPv2 bilingual agent code
stacks.o stacks_68.o	Per-process stack creation, manipulation, and monitoring routines
sum.o	Checksum support
util.o	Miscellaneous utility routines, including time-related services and case conversion

CPU Profiling

C.1 Overview: CPU Profiling

This chapter describes a low-overhead method of CPU profiling, which allows you to determine what the CPU spends its time doing. CPU profiling is quite useful during code development to help focus attention on the areas that require optimization. It is also useful in the field and the lab to help track down performance problems.

The method of CPU profiling described in this appendix consists of two parts: sampling the CPU and creating a graph of CPU usage. First, this method samples the location of the processor every 4 milliseconds (2 50times per second). Each one of these time increments is referred to as a *tick*. The sampling result is a histogram of code coverage. Because the sampling is done from the nonmaskable interrupt (NMI), the profiler tracks the execution of interrupt routines and tracks within sections of code where interrupts have been excluded.

Second, a postprocessing program takes the profile data and a symbol table and produces a graph of CPU usage along with a ranking of the most processor-intensive modules and procedures.

C.2 How CPU Profiling Works

C.2.1 Define Profile Blocks

To profile a section of code, you define one or more *profile blocks*. A profile block is a block that specifies an address range and a granularity. The address range specifies the range of code to be profiled.You determine the range manually, usually by looking it up in the symbol table. The granularity specifies the fineness of the profile, down to single instructions. The finer the granularity, the more memory the profiler needs in order to run.

C.2.2 Profile Block Bins

Each profile block is represented by a set of bins. The number of bins depends on the size of the block and its granularity. When a CPU location is sampled, if it lies within a profile block, the corresponding bin is incremented.

Profile blocks can overlap. If the CPU is running in a location that lies within multiple profile blocks, the appropriate bin for each block is incremented.

C.2.3 Tracking Tic

The profiler keeps track of the total number of ticks, regardless of whether the CPU is caught in a profile block. This allows the postprocessor to calculate absolute CPU percentages, regardless of how much of the code is being profiled.

C.2.4 Overhead

The overhead for CPU profiling is generally minimal. If CPU profiling is disabled, the cost of the profiling system is the cost of a single compare instruction in the NMI thread. If CPU profiling is enabled, it requires a single procedure call and roughly 20 instructions (on a 680*x*0 processor) in the NMI thread per profile block. The overhead increases significantly if you run CPU profiling in CPUHOG mode, which is described in the section "Special Modes."

Each profile bin requires 4 bytes of memory. To determine the number of bins required to support a profile block, divide the block size (end size minus start size) by the granularity.

C.2.5 Special Modes

Normally, CPU profiling runs continuously. However, it can also run in the following special modes:

- Task mode. In this mode, the profiler counts a CPU tick only if one of a particular set of processes is running. One use of this mode is to determine which process is spending excessive time in a particular portion of common code. If you do not use task mode in this case, the profiler reports only that a procedure is called a lot, but does not report who the caller is.
- Interrupt mode. This mode is similar to task mode, but it counts ticks only if interrupts are being excluded to some degree, either because an interrupt routine is running or because interrupts are being temporarily excluded by a process. You can run task and interrupt modes simultaneously.
- CPUHOG mode. This mode is useful for tracking down processes that use excessive amounts of CPU. When the process scheduler detects that a process has held the CPU for an unreasonable length of time (currently the default is 2 seconds), the scheduler declares a CPUHOG event. CPUHOG mode zeros all profile bins each time a process is given control and stops profiling when a CPUHOG condition is detected. CPUHOG mode provides a snapshot of where the CPU was spending its time when the hogging process was running.

Note CPUHOG mode incurs significant overhead because the blocks are zeroed before each process is invoked. Keep the number and size of the profile blocks to a minimum, or you will bring the system to its knees.

You can run CPUHOG and task modes simultaneously.

C.3 Caveats about Using CPU Profiling

Nothing is perfect in this universe, and the CPU profiler is definitely less than perfect. Do not blindly accept what it tells you. You need to understand how it determines what it tells you.

The profiler is stochastic in nature. Although 250 samples per second might seem like a lot, the system can do a lot in 4 milliseconds. In general, the longer you collect data, the more accurate the profiling will be.

The profiler's sample is biased, which might cause problems. In particular, the profiler is synchronized with the timer system. For instance, if a process is waiting on a sleeping timer, the profiler is guaranteed never to catch the CPU in that process unless the system is so loaded that the process latency is at least 4 milliseconds. However, if you are running the profiler to determine why a system is extremely loaded, the CPU profiler will generally catch the perpetrator.

If you use a granularity larger than one instruction, the postprocessor cannot accurately resolve module and procedure boundaries because a single bin can span a boundary. To avoid this problem, sample at a fine granularity over a small range. It might take several profiling runs to do this—first profile at coarse granularity over a wide range, then zoom into a finer granularity.

C.4 Use the CPU Profiler

To use the CPU profiler, follow these steps:

Step Configure the profiler to capture the data that you want. Specify a mode appropriate to the problem you are trying to solve.

Step After you have captured the data, use Telnet to connect to the system and direct the output of **show profile terse** to a file on a UNIX system. Do *not* attempt to do this via the console port. The console port is slow and does not obey flow control, so data will be lost. Be aware that **show profile terse** disables `automore()` processing, so once the profile data starts to flow it will not stop until it has all been dumped.

Step Pass the captured file, along with a symbol table matching the image running in the system, to the postprocessing program.

Step Scratch your head and mutter.

Step Repeat Ste p1 through Step 4.

C.5 Configure the Profiler

Most commands for running the CPU profiler are EXEC commands. However, you can issue the command that defines profile blocks by EXEC or you can include it in your configuration file. Including it in a file is handy for tracking problems that are hard to reproduce.

All the CPU profiler commands are hidden.

C.5.1 Create a Profile Block and Enable Profiling

To create a profile block and enable profiling, use the **profile** EXEC command. To delete a profile block, use the **no** form of this command.

profile *start end increment*
no profile *start end increment*

start is the starting address and *end* is the ending address of an address range that specifies the range of code to be profiled. You determine the range manually, usually by looking it up in the symbol table. Specify the address as hexadecimal numbers without the leading 0x.

increment specifies the granularity of the profiling. The granularity specifies the fineness of the profile, down to single instructions. The finer the granularity, the more memory the profiler needs in order to run. An increment of two on a 680*x*0-based machine or four on an R4000-based machine provides per-instruction granularity. Specify the granularity as hexadecimal numbers without the leading 0x.

Creating a profile block enables profiling, and the bins start to increment immediately.

By default, the CPU profiler is disabled.

C.5.2 Delete a Profile Bloc

To delete a profile block, use one of the following EXEC commands:

no profile *start end increment*

unprofile {*start end increment* | **all**}

C.5.3 Stop Profiling

To stop CPU profiling, use the **profile stop** EXEC command:

profile stop

When profiling stops, all data in the profile bins is preserved.

C.5.4 Restart Profiling

To restart CPU profiling after you have stopped it with the **profile stop** command, use the **profile start** EXEC command:

profile start

When profiling restarts, all new data is appended to that in the existing profile bins.

C.5.5 Zero Profile Blocks

To zero all profile blocks, use the **clear profile** EXEC command:

clear profile

C.5.6 Enable Task and Interrupt Modes

To enable task and interrupt modes, use the **profile task** EXEC command:

profile task [**interrupt**] [*pid1*] [*pid2*] ... [*pid10*]

The parameters *pid1* through *pid10* are the process IDs shown in the **show process** command.

Executing the **profile task** command does not affect the status of the profiler, that is, whether it is running or stopped.

C.5.7 Disable Taskand Interrupt Modes

To disable task and interrupt modes, use the **unprofile task** EXEC command:

unprofile task

Executing the **unprofile task** command does not affect the status of the profiler, that is, whether it is running or stopped.

C.5.8 Enable CPUHOG Profiling

To enable CPUHOG mode, use the **profile hogs** EXEC command:

profile hogs

This command enables profiling, clearing all bins before executing each process. Profiling is halted when the first CPUHOG event occurs, effectively executes a **profile stop** command immediately after the CPUHOG condition is detected.

To restart the profile in CPUHOG mode, issue another **profile hogs** command.

C.5.9 Display Profiling Information

To display CPU profiling information, use the **show profile** EXEC command:

show profile [**detail** | **terse**]

Specify the **detail** argument to format the contents of the profile bins nicely. This formatting is of dubious value. Specify the **terse** argument to format the contents of the bins in a form suitable for postprocessing. The **show profile terse** does *not* obey automore processing, so use this command only when you really mean it.

C.6 Process the Profiler Output

The postprocessing program, `profile`, reads CPU profiling statistics and formats them in a reasonably useful, albeit crude, fashion. It is currently located in the `/csc/tools/profile` directory. This directory will change when we find a good home for the program.

To invoke the postprocessing program, use the **profile** command:

profile *symbol-file data-file* [*magnification* [*width*]]

symbol-file is the symbol table file that corresponds to the image running in the system.

data-file contains the captured output of the **show profile terse** command. The file may contain the extraneous noise that is unavoidable when capturing a terminal session. The postprocessor automatically finds the data it needs.

magnification is the magnification factor for the histogram. The default is 1, which means that the histogram is scaled such that a histogram line that fills the width of the screen is equal to 100% CPU load. A magnification of 2 means that the screen width corresponds to 50% CPU load, and so on. A magnification value of about 10 is usually a good value to start with when examining histogram data.

width is the screen width. The default is 80 columns.

The `profile` program first produces an annotated histogram of CPU utilization, scaled according to the selected magnification and screen width. Long lines are truncated, and `-->` is added to the end of truncated lines. Lines of zero length line are not displayed. This means that as you turn up the magnification, more and more lines appear in the histogram.

The histogram is annotated with module and procedure names. If the granularity is so coarse that a profile block crosses procedure boundaries, the first procedure is displayed before the histogram line, and each of the others follows the histogram line. If the block crosses module boundaries, the

same effect occurs. However, if a block completely subsumes a module, none of the component procedures are listed. Procedure names include their offset into their parent module, making it easy to correlate the histogram with an object listing.

After the histogram, the `profile` program lists the top 25 modules and 100 procedures, in terms of CPU utilization, along with their utilization percentage. If a block crosses procedure or module boundaries, all CPU use in the block is charged to the first procedure or module.

The profiling support allows multiple blocks—and even overlapping blocks—to be profiled simultaneously. The output is produced for each block separately. CPU percentages are absolute. Even if only a small section of the system is profiled, the percentages reflect total CPU utilization.

The `profile` program is most useful when analyzing a narrow range of code at very fine granularity A block resolution of 2 bytes allows you to have instruction-by-instruction analysis capabilities, but this obviously requires lots of memory on the router

A reasonable alternative is to run one block at fairly coarse granularity (say, 256 bytes) covering the whole system, then focus in on the trouble spots.

APPENDIX D

Older Version of the Scheduler

With Cisco IOS Release 11.0, the scheduler was significantly redesigned. However, elements of the previous scheduler design—especially, how the scheduler processes queues—are still supported in releases prior to Release 11.0. These elements of the older scheduler design have been eliminated from Releas e11.3 of the Cisco IOS code.

This chapter describes the features and API functions that were peculiar to the scheduler prior to Release 11.0. You should use these features and functions only when maintaining existing Cisc oIOS code in releases prior to Release 11.0. When writing code for Cisco IOS Releases 11.0 and later, you should use the features and functions described in the "Scheduler" chapter in this manual.

When it is necessary to compare the two versions of the scheduler, the redesigned code is referred to as the *new* scheduler and the previous scheduler design is referred to as the *old* scheduler

This chapter describes only those portions of the old scheduler that differ from the new scheduler; it does not provide a complete description of the old scheduler. For this, you must use this chapter in conjunction with the "Scheduler" chapter in this manual.

D.1 How a Process Stops

In the new scheduler, a process stops executing by killing itself. The `main` routine of a process must explicitly call the `process_kill()` function; it cannot just execute a `return` statement. The latter is considered an error condition and is protected against. When processes are no longer needed—for example, when a protocol is unconfigured—they should clean up after themselves and exit.

Many processes written with older versions of the scheduler code do not exit when they are no longer needed and thus waste system resources. These older processes are the exception, not the norm.

D.2 Queues and Process Priorities

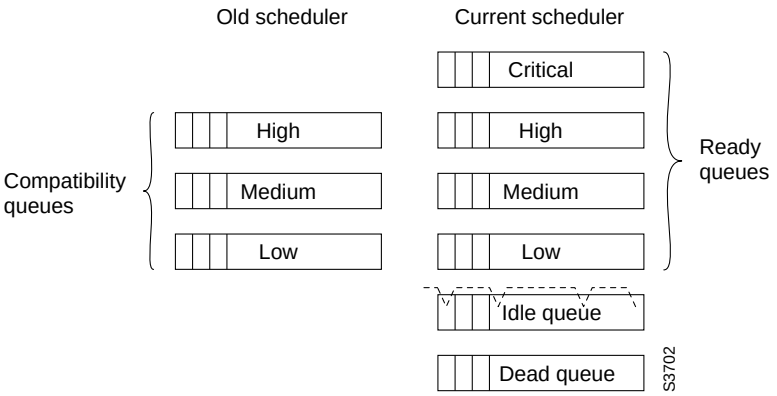
D.2.1 Scheduler Queues

In addition to the ready, idle and dead queues, Releases 11.0, 11.1, and 11.2 of the Cisco IOS software supports a fourth type of queue, the compatibility queue, for compatibility with the old scheduler. Compatibility queues are similar to the new scheduler's ready queues.

D.2.1.1 Comparison of New and Old Scheduler Queues

Figur eD-1 compares the new scheduler queues to the old scheduler queues. The dotted line above the idle queue shows that some of the items on the idle queue are also threaded onto a list. This is the list of items whose wakeup reasons include or consist solely of a timer expiration.

Figure D-1 New and Old Scheduler Queues



D.2.1.2 Compatibility Queues

Compatibility queues were used by the old scheduler. These queues are available in Releases 11.0, 11.1, and 11.2 for backward compatibility only.

A compatibility queue is for processes that may be ready to run, but that may also be waiting for an arbitrary event to occur. This arbitrary event is detected by a code fragment that is executed within the scheduler context once for every pass of the queue.

Compatibility queues can handle processes of high, medium, and low priority. They do not provide a critical-priority queue.

D.2.1.3 Idle Queue

The idle queue is for processes that are waiting for an event to occur before they can execute. In the old scheduler model, the scheduler performed background polling to determine whether processes needed to be awakened. As an example, the VINES process polls the queue at every pass of the scheduler

D.2.2 Operation of Scheduler Queues

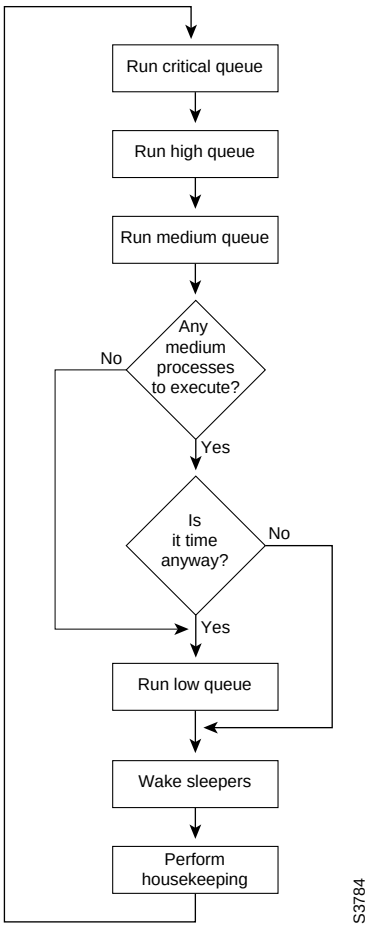
For Release 11.0, 11.1, and 11.2, which have both the old and new schedulers, the old and new scheduler queues are processed in parallel. The new scheduler is similar to the old model. In both models, the high-priority queues (and, for the new scheduler, the critical-priority queues) are processed multiple times for each pass at the medium- and low-priority queues.

D.2.2.1 Overall Scheduler Queue Operation

The overall scheduler queue operation is as follows. Fi gureD-2 illustrates this operation.

1. Run each process in the critical-priority list.
2. Run each process in the high-priority list.
3. Run each process in the medium-priority list.
4. Possibly run each process in the low-priority list.
5. Wake sleeping processes. All processes are threaded via managed timers. The scheduler checks the parent timer for expiration time and moves expired processes to the appropriate run queues.
6. Perform housekeeping operations. These include computing CPU loads and busy times, performing postmortem analysis on processes, performing “scheduler-interval” processing, and spinning a random-number generator.

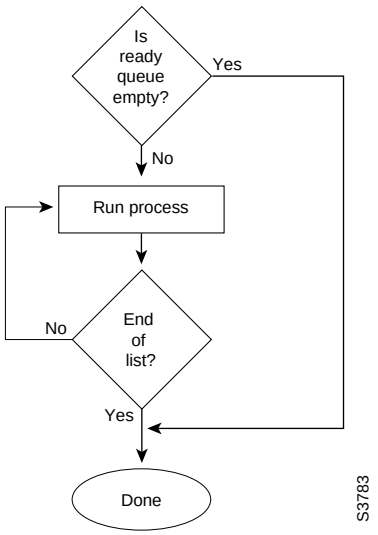
Figure D-2 Overall Scheduler Queue Operation



D.2.2.2 Critical-Priority Scheduler Queue Operation

Processes on critical-priority queue are handled by the scheduler immediately, as illustrated in Figur eD-3.

Figure D-3 Critical-Priority Scheduler Queue Operation



D.2.2.3 High-Priority Scheduler Queue Operation

The operation of the high-priority queue is as follows. Figur eD-4 illustrates this operation.

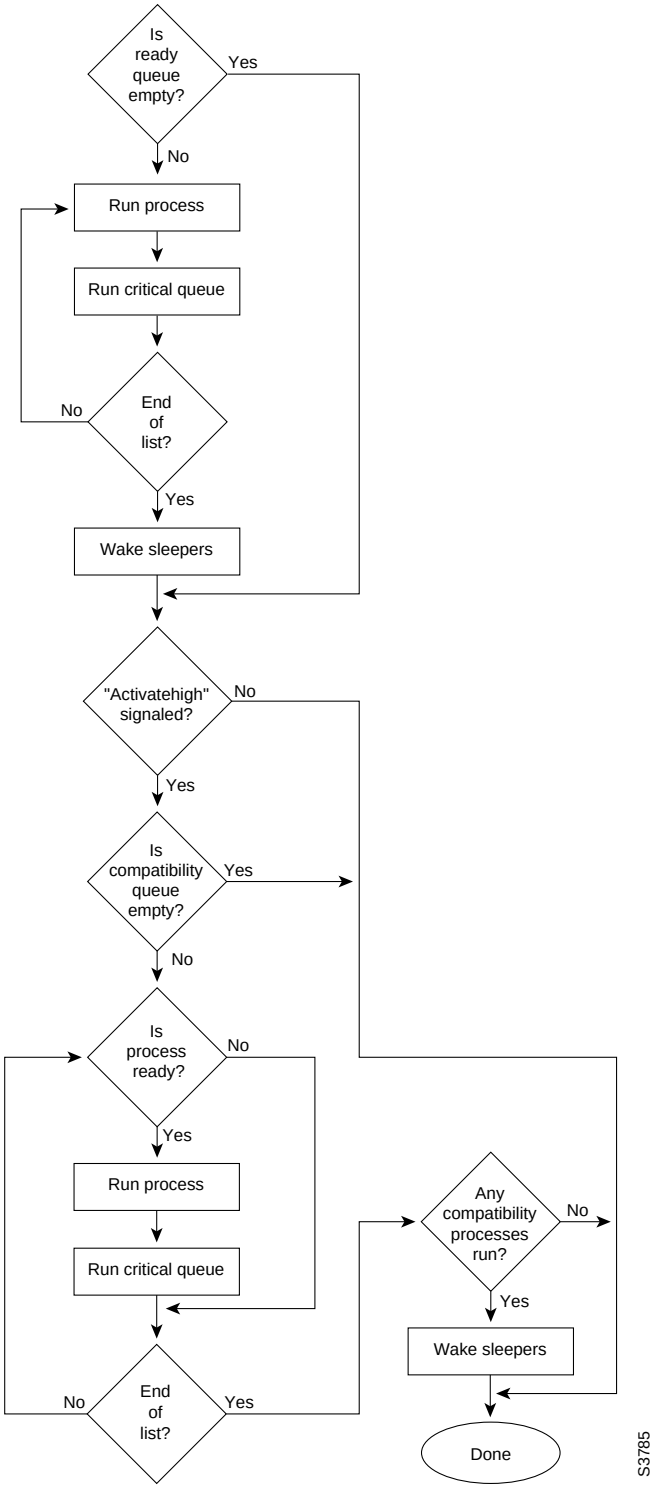
Check the Ready Queues

1. For each process in the list:
   - Run the process.
   - Run each process in the critical-priority list.
2. Wake sleeping processes.

Check the Compatibility Queues

3. Test the “activatehigh” flag.
4. For each process in the list:
   - Test if the process is waiting for an event.
   - Run the process.
   - Run each process in the critical-priority list.
5. Wake sleeping processes.

Figure D-4 High-Priority Scheduler Queue Operation



D.2.2.4 Medium- and Low-Priority Scheduler Queue Operation

The operation of the medium-priority queue is as follows. Figur eD-5 illustrates this operation.

Check the Ready Queues

1. For each process in the list:
   - Run the process.
   - Run each process in the critical-priority list.
   - Run each process in the high-priority list.
2. Wake sleeping processes.

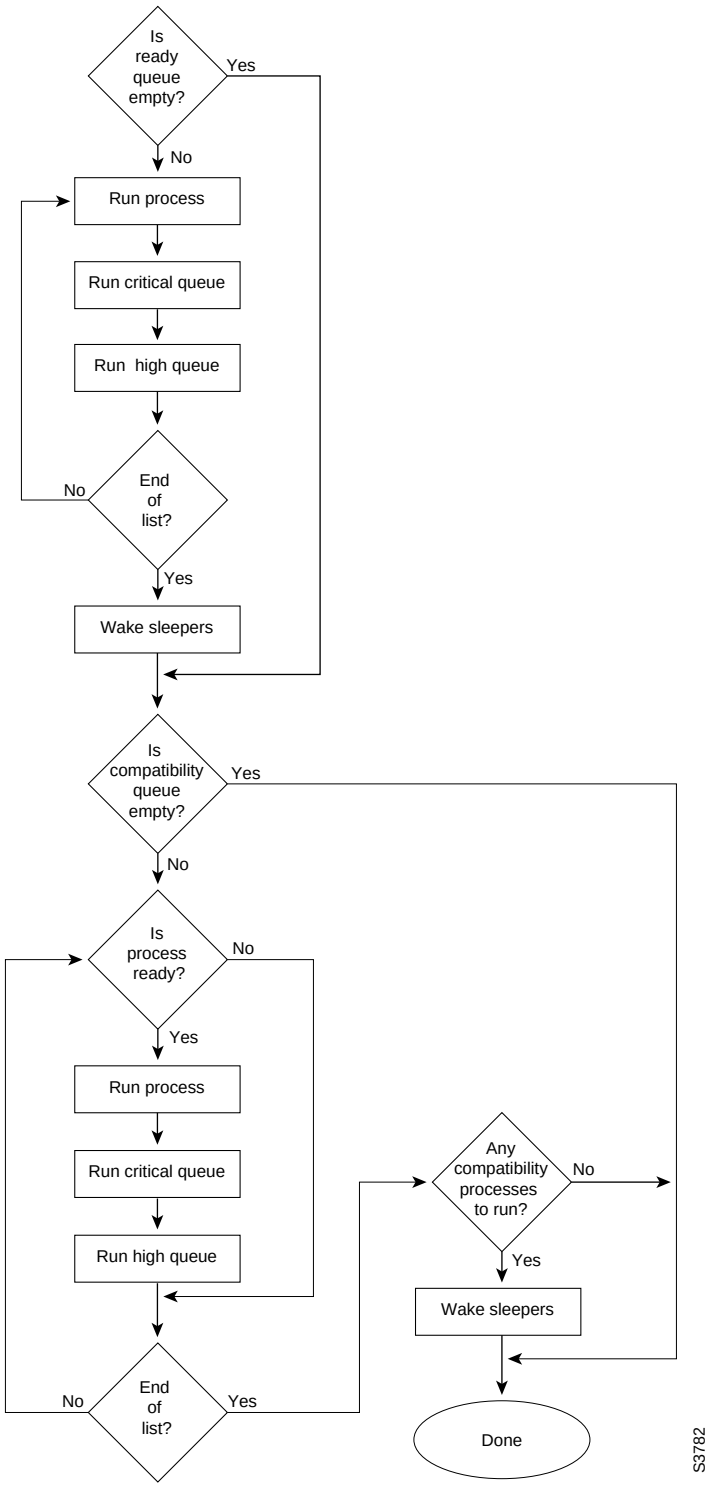
Check the Compatibility Queues

3. For each process in the list:
   - Test if the process is waiting for an event.
   - Run the process.
   - Run each process in the critical-priority list.
   - Run each process in the high-priority list.
4. Wake sleeping processes.

The operation of the low-priority queue is identical to that of the medium-priority queues and is shown in Figure D -5. Processes in the low-priority queue are executed under the following conditions:

- When no medium-priority processes are executed on this pass of the scheduler
- Whenever there have been 20 passes through the medium-priority list since the last pass through the low-priority list

Figure D-5 Medium- and Low-Priority Scheduler Queue Operation



D.3 Functions in the Old Scheduler

While support for some functions in the old scheduler was maintained in Releases 11.0, 11.1, and 11.2 code, you should avoid using them in these versions of the code. As of Releas e11.3, support for these functions has been removed.

Table D-1 list some obsolete old scheduler functions and their equivalent functions in the new scheduler. The following sections discuss some of the functions in the old scheduler.

Table D-1 Comparison between Old and New Scheduler Functions

Old Scheduler Function	Equivalent New Scheduler Function
`adisms()`	`process_sleep_until()`
`cfork()`	`process_create()`
`check_suspect()`	`process_may_suspend()`
`edisms()`	`process_wait_for_event()`
`pdisms()`	`process_sleep_periodic()`
`process_is_high_priority()`	—
`process_set_priority()`	`process_create()`
`s_kill()`	`process_kill()`
`s_suspect()`	`process_suspend()`
`s_tohigh()`	`process_create()`
`s_tolow()`	`process_create()`
`tdisms()`	`process_sleep_for()`

D.4 cfork() (obsolete)

To create a new process, call the `cfork()` function.

```
#include "sched.h"
pid_t cfork(forkproc (*padd), long pp, int stack, char *name, int ttynum);
```

Classification

Function

Input Parameters

padd	Starting address of the process to execute.
pp	Parameter to the process.
stack	Size of the process stack in *words*. A value of 0 uses the default stack size.
name	Textual name of the process as it should appear in all output.
ttynum	Controlling terminal number for this process. Processes that do not use a controlling terminal should set this parameter to 0.

Output Parameters

None

Return Type

`pid_t`

Return Values

pid	Process identifier of the newly created process.
`NO_PROCESS`	A new process could not be created.

Usage Guidelines

The `cfork()` function was available in Cisco IOS releases prior to Release 11.0. It is no longer supported. Call the `process_create()` function instead.

Note You specify the stack size in words, not in bytes.

Side Effects

`cfork()` creates a new process and places it into the normal priority run queue. Priority is set with `process_set_priority()` [another obsolete function]. The process will not begin executing until the current process is dismissed.

Related Functions

```
process_kill()
process_set_arg_num()
process_set_arg_ptr()
process_set_ttynum()
process_set_ttysoc()
```

D.5 edisms() (obsolete)

To suspend a process by putting it to sleep until some arbitrary event occurs, call the `edisms()` function.

```
#include "sched.h"
void edisms(uchar *test_routine, ulong pp);
```

Classification

Function

Input Parameters

test_routine	Arbitrary code fragment that is executed by the scheduler to determine whether the process should continue sleeping. This routine should return TRUE is the process should continue sleeping, and FALSE if the process should wake up.
pp	Parameter to the test function.

Output Parameters

None

Return Type

`void`

Return Values

None

Usage Guidelines

The `edisms()` function was available in Cisco IOS releases prior to Release 11.0. It is no longer supported. Call the `process_wait_for_event()` function instead.

This function introduces a large amount of overhead into the scheduler, because the *test_routine* must be executed at each pass of the scheduler queues.

Related Functions

`process_sleep_for()`
`process_sleep_on_timer()`
`process_sleep_periodic()`
`process_sleep_until()`
`process_wait_for_event()`

D.6 process_is_high_priority( ) (obsolete)

To retrieve the argument to the `main` routine of a process, call the `process_is_high_priority()` function.

```
#include “sched.h”
static inline boolean process_is_high_priority(void);
```

Classification

Function of class `process_get_info`

Input Parameters

None

Output Parameters

None

Return Type

`boolean`

Return Values

`TRUE`	The executing process is a high-priority process.
`FALSE`	The executing process is not a high-priority process.

Usage Guidelines

The `process_is_high_priority()` function is provided for backward compatibility only. Do not use it in any new code. Instead, write all new code to be priority independent.

D.7 process_set_priority( ) (obsolete)

To set the priority of the process that is currently running, call the `process_set_priority()` function.

```
#include “sched.h”
static inline boolean process_set_priority(int priority);
```

Classification

Function of class `process_set_info`

Input Parameter

priority	New priority for this process. This value can be one of the following: • PRIO_CRITICAL • PRIO_HIGH • PRIO_NORMAL • PRIO_LOW

Output Parameters

None

Return Type

boolean

Return Values

TRUE	The set call succeeded.
FALSE	The set call failed.

Usage Guidelines

The process_set_priority() function is provided for backward compatibility only. All newly written code should supply the process priority when the process is created by calling process_create().

Related Functions

process_create()
process_get_priority()
process_set_arg_num()
process_set_arg_ptr()

D.8 s_tohigh() (obsolete)

To change the currently executing process to run in the high priority queue, call the s_tohigh() function.

```
void s_tohigh(void);
```

Classification

Function

Input Parameters

None

Output Parameters

None

Return Type

`void`

Return Values

None

Usage Guidelines

The `s_tohigh()` function was available in Cisco IOS releases prior to Release 11.0. It is no longer supported. Call the `process_create()` function instead.

Related Function

`process_create()`

D.9 s_tolow( ) (obsolete)

To change the currently executing process to run in the low priority queue, call the `s_tolow()` function.

```
void s_tolow(void);
```

Classification

Function

Input Parameters

None

Output Parameters

None

Return Type

`void`

Return Values

None

Usage Guidelines

Do not use this function. Instead, set the process priority during the call to the `process_create()` function.

The `s_tolow()` function was available in Cisco IOS releases prior to Release 11.0. It is no longer supported. Instead, set the process priority during a call to the `process_create()` function.

Related Function

`process_create()`

APPENDIX E

Glossar

A

Abstract Syntax Notation 1
See ASN.1.

agent
In SNMP, a process on a managed system that answers request from a manager

ASN.1
Formal language used by SNMP.

asynchronous notification
Proactive message from an agent to manager providing information to the manager.

AVL tree
Balanced search trees named for Adel'son-Vel'skii and Landis, who introduced this class of balanced search trees. Balance is maintained in an AVL tree by use of rotations.

B

base class
See class.

binary tree
Data structure suitable for storage and keyed retrieval of information.

bit field
32-bit quantity in which each of the individual bits has significance.

boolean
Memory location that holds one of two values, TRUE or FALSE (that is, the value 1 or 0, respectively). See also managed boolean.

buffer pools
Grouping of buffers that allows them to be managed. Buffer pools hold buffers that are the same size and have the same properties. Buffer pools are based on the generic pool manager.

C

CANA
Cisco Assigned Numbers Authority. Group that assigns MIB branch numbers within the Cisco branch to Cisco developers.

child timer
See leaf timer.

chunk
Large block of memory that is allocated by the Cisco IOS code and then subdivided into smaller chunks. Use chunks to reduce the memory overhead when allocating a large number of small data structures. Chunks are managed by the chunk manager.

chunk manager
Code that manages chunks.

class
[object-oriented programming] a template definition of the methods and variables in a particular kind of object. Thus, an object is a specific instance of a class; it contains real values instead of variables. The class is one of the defining ideas of object-oriented programming. These are some of the important ideas about it:

- A class can have subclasses (also called derived or child classes) that can inherit all or some of the characteristics of the class. In relation to each subclass, the class becomes the superclass (also called base or parent class).
- Subclasses can also define their own methods and variables that are not part of their superclass.
- The structure of a class and its subclasses is called the class hierarchy.

cookie PROM
PROM that holds all the information that is unique to a particular physical platform, such as the chassis serial number, the MAC addresses reserved for the chassis to use, the vendor (for OEM hardware), and the type of interfaces present (for nonmodular platforms).

CPU exception
Error that occurs when the executing thread of control attempts to perform an undefined operation, such as accessing an invalid address in memory or dividing by zero.

cruft
Worthless rubbish, usually used in terms of something being superfluous but attached to a valued object.

CSB
Console status block.

CSB objects
In the parser, a generic way to reference parser variables.

CxBus
Cisco Extended Bus. Data bus for interface processors on Cisco 7000 series routers that operates at 533 Mbps. See also SP (switch processor).

D

dead queue
Scheduler queue for processes that have exited, but on which the schedule has not yet performed a postmortem analysis.

demand paging
A kind of virtual memory where a page of memory will be paged in if an attempt has been made to access it and it is not already present in main memory.

direct queue
Singly linked list in which the first longword of the data structure is reserved for linking together the items in the list.

doubly linked list
A linked list with an embedded pointer block containing forward and backward pointers. Multiple pointer blocks can be embedded in the same data structure, allowing it to be on multiple doubly linked lists at the same time.

E

entity
In IPCs, a procedure or routine, such as a process, executing code, or a module, that makes use of IPC services.

epoch
Instantaneous location in time.

exception
Error that occurs in the execution of the Cisco IOS code. The error is converted into a signal before being offered to the software for exception handling.

G

GAS
GNU Assembler, supported by Cygnus.

GCC
GNU compiler, supported by Cygnus.

GDB
GNU debugger, supported by Cygnus.

H

heap
Memory that remains in a region after an imaged has been loaded.

I

IANA
Internet Assigned Numbers Authority. Group that assigns MIB branch numbers to private enterprises.

IDB
Interface descriptor block. There are several types of IDBs, including hardware IDBs and software IDBs. They are structures that describe the hardware and software view of an interface.

IDB subblock
Area of memory that is private to an application and that is used to store private information and state variables that the application wants to associate with an IDB or interface.

idle queue
Schedule queue for processes that are waiting for an event to occur before they can execute. The event must be one of a set of events explicitly listed by the process.

in-band signaling
Transmission within a frequency range normally used for information transmission. Contrast with OOB (out-of-band signaling).

indirect queue
Singly linked list with queuing blocks. These functions have no requirements regarding the format of the data structure.

inform
Type of asynchronous notification in which are acknowledged datagrams that are set from one manager process to another.

inheritance
[object-oriented programming] the concept that when a class of objects is defined, any subclass that is defined can inherit the definitions of one or more general classes. This means for the programmer that an object in a subclass need not carry its own definition of data and methods that are generic to the class (or classes) of which it is a part. This not only speeds up program development; it also ensures an inherent validity to the defined subclass object (what works and is consistent about the class will also work for the subclass).

instance identifier
Designation of a specific occurrence of an object in the MIB tree. Also called an object identifier.

Internet Network Management Framework
Framework on which SNMP is based. It defines a model in which a managing system called a manager communicates with a managed system, which runs an agent.

interval tree
Variation of an RB tree in which the key is a range instead of a single number.

IPC
Interprocess Communications.

J

jitter
Method of randomizing an expiration time within set limits.

L

leaf timer
In managed timers, a timer that has no dependency on any other timer. Leaf timers are grouped together under a parent timer, and the parent timer expires at the earliest leaf expiration time. Also called a child timer.

link point
Locations in the parse tree where new commands can be dynamically added. They are used to allow the partial loading of commands.

list manager
Set of functions for manipulating doubly linked lists.

M

managed boolean
Boolean that can wake up a process or processes whenever the value of the boolean is set to `TRUE` (that is, the value 1). Also referred to as a watched boolean.

managed queue
Queue that can be managed by the scheduler. The process associated with the queue is awakened any time a new element is added to the queue. Also referred to as a watched queue.

managed semaphore
Data structure that contains a simple semaphore and all the other information necessary for the semaphore to be used as a scheduler wakeup condition. Also referred to as a watched semaphore. See also semaphore, simple semaphore.

managed timers
Timers that augment passive timers by allowing you to group timers together, thus allowing you to conveniently and efficiently manipulate a large number of timers. A parent timer is used to represent a group of leaf (child) timers.

Management Information Base
See MIB.

manager
In SNMP, an application running on a managing system that requests information from an agent.

memory management unit
See MMU.

memory pools
Pools used to manage heaps.

message
In the scheduler, a simple interprocess communication (IPC) mechanism that works on a single processor. It allows two processes to communicate.

In IPCs, the basic unit of communication exchanged between entities.

method
[object-oriented programming] programmed procedure that is defined as part of a class (*see*) and included in any object [*see*] of that class. A class (and thus an object) can have more than one method. A method in an object can only have access to the data known to that object, which ensures data integrity among the set of objects in an application. A method can be re-used in multiple objects.

MIB
Management Information Base. Abstract database description that defines all the information about a managed system that a manager can view or modify.

MMU
hardware device used to support virtual memory and paging by translating virtual addresses into physical addresses.

MTU
Maximum Transmission Unit. Maximum size of any frame that can be transmitted on a particular media.

multicast ports
In IPCs, an aggregation of ports referenced as a single port so that messages can be transmitted from one source to multiple destinations.

N

Network Time Protocol (NTP
See NTP.

NTP
Network Time Protocol. 1. Cisco IOS time protocol that maintains the system clock to a very high degree of accuracy, adjusting the clock frequency to correct for the otherwise unavoidable drift caused by systematic errors in the clock hardware. 2. Protocol designed to synchronize timekeeping among a set of distributed timer servers and clients. NTP runs over the User Datagram Protocol (UDP) and the Internet Protocol (IP).

O

object
1. leaf in the MIB tree. Sometimes also called a variable. 2. [object-oriented programming] Objects are the things you think about first in designing a program and they are also the units of code that are eventually derived from the process. In between, each object is made into a generic class of object and even more generic classes are defined so that objects can share models and reuse the class definitions in their code. Each object is an instance of a particular class or subclass with the class's own methods or procedures and data variables. An object is what actually runs in the computer.

object identifier
Unique name of a data object or other registration point in a MIB tree.

OID
Object identifier.

oneshot notification
Awakening a process only the first time a scheduler object, such as a boolean, changes.

OOB
(out-of-band signaling) Transmission using frequencies or channels outside the frequencies or channels normally used for information transfer. Out-of-band signaling is often used for error reporting in situations in which in-band signaling can be affected by whatever problems the network might be experiencing. Contrast with in-band signaling.

out-of-band signaling
See OOB.

P

page fault
[virtual memory] an access to a page (block) of memory that is not currently mapped to physical memory.

paging
technique for increasing the memory space available by moving infrequently-used parts of a program's working memory from RAM to a secondary storage medium, usually disk. The unit of transfer is called a page. See also page fault, MMU, virtual memory.

parent timer
In manager timers, a timer that represents a group of leaf (child) timers. This timer always expires at the earliest expiration time of any of its child timers.

passive timers
Timers that note the current value of the system clock and record the value either as it is or after adding a delay value.

PID
Process identifier.

PIM
Protocol Independent Multicast. Multicast routing architecture that allows the addition of IP multicast routing on existing IP networks. PIM is unicast routing protocol independent and can be operated in two modes: dense mode and sparse mode. See also PIM dense mode and PIM sparse mode.

PIM dense mode
One of the two PIM operational modes. PIM dense mode is data-driven and resembles typical multicast routing protocols. Packets are forwarded on all outgoing interfaces until pruning and truncation occurs. In dense mode, receivers are densely populated, and it is assumed that the downstream networks want to receive and will probably use the datagrams that are forwarded to them. The cost of using dense mode is its default flooding behavior. Sometimes called dense mode PIM or PIM DM. Contrast with PIM sparse mode. See also PIM.

PIM sparse mode
One of the two PIM operational modes. PIM sparse mode tries to constrain data distribution so that a minimal number of routers in the network receive it. Packets are sent only if they are explicitly requested at the RP (rendezvous point). In sparse mode, receivers are widely distributed, and the assumption is that downstream networks will not necessarily use the datagrams that are sent to them. The cost of using sparse mode is its reliance on the periodic refreshing of explicit join messages and its need for RPs. Sometimes called sparse mode PIM or PIM SM. Contrast with PIM dense mode. See also PIM and RP (rendezvous point).

pool cache
Lookaside list of free items that can be accessed quickly.

port
In IPCs, a communications end point.

port identifier
In IPCs, a 32-bit integer that uniquely references a communications end point.

port name
In IPCs, a textual name of a port that is registered with the local seat manager and is associated with the port's identifier.

port table
In IPCs, a table that contains information about local ports available to users.

prepaging
technique whereby the operating system in a paging virtual memory multitasking environment loads all pages of a process's working set into memory before the process is restarted.

priority
Order in which the scheduler executes processes. Processes can run at one of the following priority levels: critical, high, medium, or low.

process
Roughly the equivalent of a *thread* in computer science terminology. A Cisco IOS process consists of a set of processor registers and a stack area.

process state
Current activity of a process. It can be one of the following: running, suspended, ready to run, waiting for event, sleeping, hung, or dead.

Q

queue
Singly linked list that is a simple data structure for maintaining a linked list of objects. It is an ordered collection of items that keeps track of the first and last objects, the current number of objects, and the maximum number of objects.

R

radix tree

RB tree
Red-Black tree. The Cisco IOS implementation is a threaded tree.

ready queue
Scheduler queue used for processes that are ready and waiting to run.

realm
In IPCs, a collection of one or more seats (that is, processors) that form a distributed system. It is within this collection that port identifiers are unique and communicating entities can be moved.

realm manager
In IPCs, global entity responsible for the set of seats making up a realm.

region
Contiguous area of the Cisco IOS address space. In its simplest form, a region is an area of memory that is described by a starting address and a size, in bytes. The Cisco IOS software uses regions to organize memory into a hierarchical and manageable scheme. Region attributes are controlled by the region manager

region class
Identifies the function for which a region os memory is used. Classes provide a method for organizing regions of memory. Examples are processor-based memory, high-speed I/O memory, and Flash memory

region manager
Code that organizes memory hierarchically so that platform-specific and driver-specific code can declare areas of memory to the kernel and the kernel can determine how much memory is installed or available in a platform.

registry
Collection of related services. In conjunction with services, registries permit subsystems to install or register callback functions, discrete values, or process IDs for a service provided by the kernel or other modules.

Remote Procedure Call
See RPC.

RP
1. Route Processor. Processor module on the Cisco 7000 series routers that contains the CPU, system software, and most of the memory components that are used in the router. Sometimes called a supervisory processor. 2. Rendezvous Point. Router specified in PIM sparse mode implementations to track membership in multicast groups and to forward messages to known multicast group addresses. See also PIM sparse mode.

Route Processor
See RP.

RPC
Remote Procedure Call. Procedure call to an application in which the actual work happens on another processor.

RSP
Route/Switch Processor. Processor module that integrates the functions of the RP and SP. (See.)

S

seat
In IPCs, a computational element, such as a processor, that can be communicated with using IPC services. A seat is where entities and ports reside.

seat manager
In IPCs, the entity responsible for the local seat.

seat table
In IPCs, a table that contains information about all seats in the IPC system.

semaphore
Memory location that is used by multiple processes to serialize their access to a set of resources. The resource can be anything, for example, Flash memory or the table of IP routes. See also managed semaphore, simple semaphore, watched semaphore.

service
Data structure that describes how a collection of one or more C functions, discrete values, or process IDs should be handled when the service is invoked by a service client. In conjunction with registries, services permit subsystems to install or register callback functions, discrete values, or process IDs for a service provided by the kernel or other modules.

service point
Actual instance of a service.

signal
See exception.

Simple Network Management Protocol
See SNMP.

simple semaphore
Single memory location that can be set or cleared by routines that function atomically. See also managed semaphore, semaphore, watched semaphore.

singly linked list
See queue.

SMI
Structure of Management Information. Defines the components of a MIB and the formal language for describing them.

SNMP
Simple Network Management Protocol. Language for communication between a managing system running a network management application and a managed system running an agent.

SNMP conceptual tables
Mechanism for defining a set of objects that appear repeatedly, indexed by some entry name.

SP
Switch Processor. Cisco 7000-series processor module that acts as the administrator for all CxBus activities. Sometimes called *ciscoBus controlle* . See also CxBus.

Structure of Management Information
See SMI.

subblock
See IDB subblock.

subsystem
Independent entry point into the Cisco IOS system code.It can be independent of the linker, or it can be freestanding code or part of code that always links and runs together. Subsystems allow images to be compiled that have the minimum of link requirements.

subsystem classes
Organized groups of subsystems that provide a sorting order that is primarily used when initializing the system software.

summer time
Daylight savings time.

Switch Processor
See SP.

system clock
Basic Cisco IOS clock. It is updated by hardware clock interrupts, advancing by an amount equal to the period of the hardware clock for each tick.

T

timers
See managed timers, passive timers.

token
Sequence of characters having a collective meaning. Characters can include an identifier, a keyword, a punctuation character, or a multicharacter operator

trap
Type of asynchronous notification in which unacknowledged datagrams that are sent by the agent to the manager.

U

UTC
Coordinated Universal Time, also known as zulu time, formerly Greenwich Mean Time (GMT). Time zone at zero degrees longitude.

V

vector
memory location containing the address of some code, often some kind of exception handler or other operating system service. By changing the vector to point to a different piece of code, it is possible to modify the behaviour of the operating system.

virtual address
memory location that is accessed by an application program that is running in a system with virtual memory. Intervening hardware and/or software maps the virtual address to real (physical) memory. During the course of execution of an application, the same virtual address may be mapped to many different physical addresses as data and programs are paged out and paged in to other locations.

virtual memory
[memory management] address space available to a process running in a system with a memory management unit (MMU).

W

watched boolean
Boolean that can wake up a process or processes whenever the value of the boolean is set to `TRUE` (that is, the value 1). Also referred to as a managed boolean.

watched queue
Queue that can be managed by the scheduler. The process associated with the queue is awakened any time a new element is added to the queue. Also referred to as a managed queue.

watched semaphore
Semaphore that contains a simple semaphore and all the other information necessary for the semaphore to be used as a scheduler wakeup condition. Also referred to as a managed semaphore. See also semaphore, simple semaphore.

Z

zone
In IPCs, a collection of seats between which communications is directly possible.

zone manager
In IPCs, the entity responsible for a group of seats that can directly communicate with each other.

Index

Index

Symbols

A

B

C

D

E

F

G

H

I

N

O

P

Q

R

S

T

U

V

W

X

Z